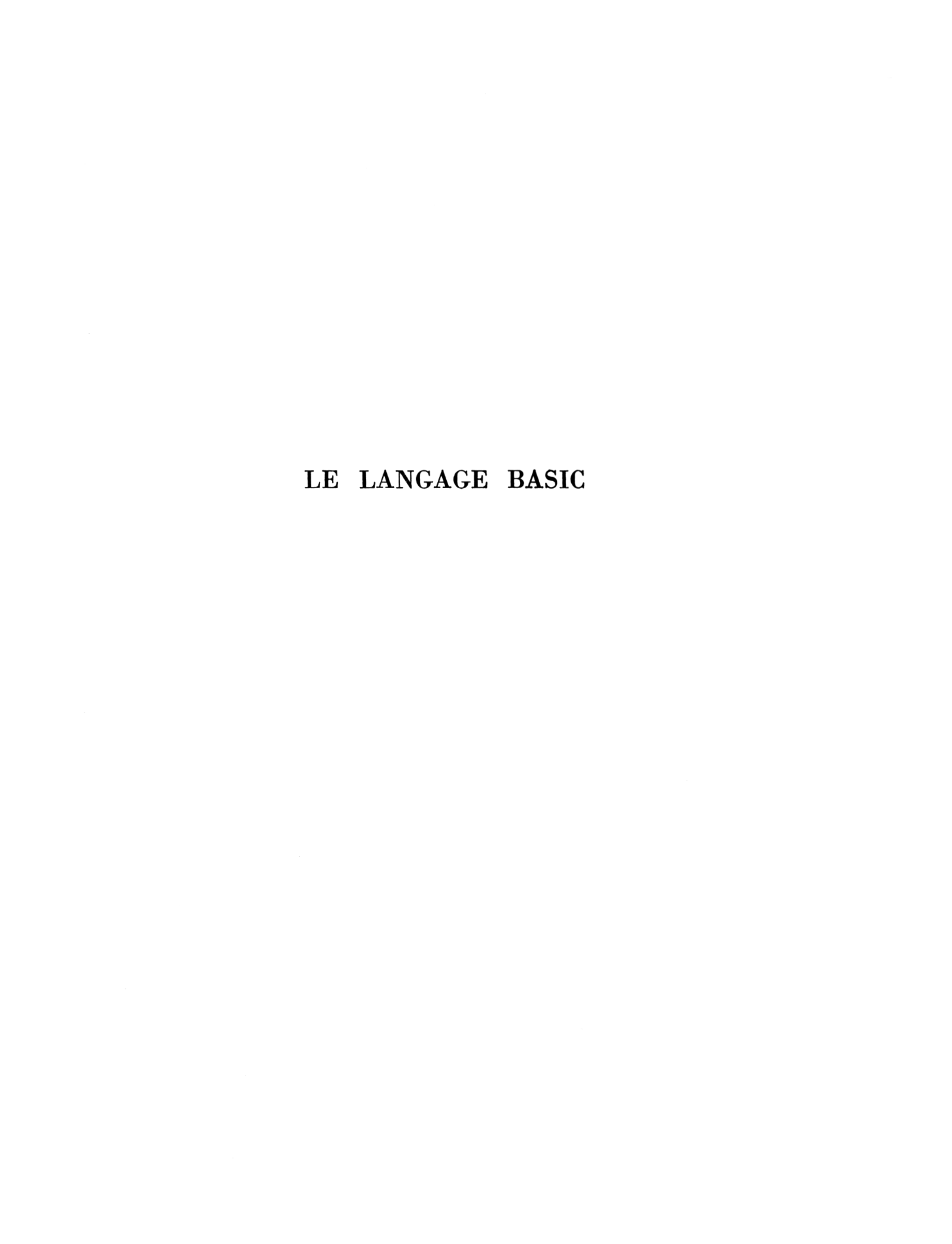


B. DRIEUX ET A.-L. LIJU

LE LANGAGE BASIC

PRESSES UNIVERSITAIRES DE FRANCE



LE LANGAGE BASIC

THÉMIS
COLLECTION DIRIGÉE PAR MAURICE DUVERGER
GESTION

BAUDOUIN DRIEUX
Chargé d'enseignement à l'Université de Lille I

ANDRÉ-LOUIS LIJU
Professeur à l'Institut de Contrôle de gestion

Le langage BASIC



PRESSES UNIVERSITAIRES DE FRANCE
108, BOULEVARD SAINT-GERMAIN, PARIS

Dépôt légal. — 1re édition : 1er trimestre 1973

AVANT-PROPOS

Le but de ce livre est de présenter un langage de programmation simple qui permet de décrire des problèmes relativement complexes sans pour autant exiger une spécialisation trop grande dans le domaine de l'informatique. Sa lecture ne nécessite aucune connaissance préalable si ce n'est la notion classique de « calcul ». La présentation du langage BASIC ne commence qu'au second chapitre. (Le premier chapitre est consacré à l'étude aussi « naïve » que possible de la notion d'ordinateur.)

Dans cette présentation, nous avons essayé de graduer les difficultés. Ainsi, les chapitres II à IV donnent les notions de base très intuitives et permettent déjà d'écrire des programmes « utiles » sans grande difficulté !

Les chapitres V à VIII abordent des notions plus complexes mais plus « riches » et, pratiquement, une grande partie des problèmes peuvent être programmés à l'issue de leur lecture.

Enfin, les chapitres IX, X et XI présentent des outils encore plus complets, qui ne deviendront vraiment indispensables que lorsque le lecteur aura acquis une certaine expérience dans le langage.

Nous avons aussi essayé de dégager de chacune des notions présentées, des concepts plus généraux qui apparaissent dans tout langage de programmation. En ce sens, nous espérons que ce livre sera pour le lecteur plutôt une introduction à la programmation à travers le langage BASIC, qu'une présentation limitée à ce langage seul.

Chapitre Premier

L'ORDINATEUR ET SES MODES DE FONCTIONNEMENT

L'informatique a connu ces dernières années un essor considérable. Ce domaine nouveau est symbolisé dans tous les esprits par ce qu'il est convenu d'appeler un ordinateur.

Le présent chapitre, introduction à l'objet de ce livre : *Le langage BASIC*, a pour but de décrire succinctement ce qu'est un ordinateur, comment il fonctionne et les différentes façons de l'utiliser.

I.1. L'ordinateur

L'ordinateur imaginé par l'homme est une machine construite à son image. Comme lui, elle peut réaliser quatre fonctions (1) :

— enregistrer des informations, telles que des nombres ou des messages, qui lui sont données ;
— garder en mémoire ces informations ;
— les traiter en effectuant par exemple des calculs sur des nombres, en comparant des messages, etc. ;
— fournir les résultats de ce traitement.

Chacune de ces fonctions est réalisée par des parties (ou organes) bien précises de l'ordinateur, à savoir :

— *les organes d'entrée :* Ce sont entre autres des machines à écrire, des lecteurs de cartes ou de rubans perforés, des lecteurs optiques...

(1) L'ordinateur, contrairement à l'homme, n'est pas capable d'imagination ni de création.

Ils permettent de fournir des informations à l'ordinateur, par exemple en tapant sur les touches d'une machine à écrire, ou en perforant des cartes et en les faisant lire par l'ordinateur ;

— *la mémoire centrale :* Schématiquement, c'est une série de « cases » dans lesquelles l'ordinateur peut garder des symboles représentant des caractères tels que des chiffres, des lettres, des signes typographiques, etc. ;

— *l'unité centrale :* C'est le « centre nerveux » de l'ordinateur. Il coordonne les transmissions d'informations entre les différents organes. Il déclenche aussi et effectue les différentes opérations que l'ordinateur est susceptible d'exécuter avec les informations qu'il possède en mémoire ;

— *les organes de sortie :* Ce sont entre autres des machines à écrire, des imprimantes, des perforateurs de cartes ou de rubans, des traceurs de courbe, des écrans cathodiques... Ils permettent de matérialiser les informations (le plus souvent les résultats d'un traitement) que l'ordinateur garde en mémoire.

A ces quatre types d'organes, il convient d'en ajouter un autre dont la nécessité apparaîtra surtout au chapitre XI : « Les mémoires auxiliaires. »

Elles permettent de garder en mémoire un très grand nombre d'informations, ce que ne peut pas faire la mémoire centrale dont la taille (le nombre de cases) est limitée pour des raisons de coût. Ces mémoires auxiliaires ont pour nom tambour, disque ou bande magnétique.

Tous ces organes, qui se trouvent physiquement dans des « armoires » différentes, sont reliés entre eux par des câbles qui permettent de coordonner leur fonctionnement et de transmettre des informations, au moyen d'impulsions électriques.

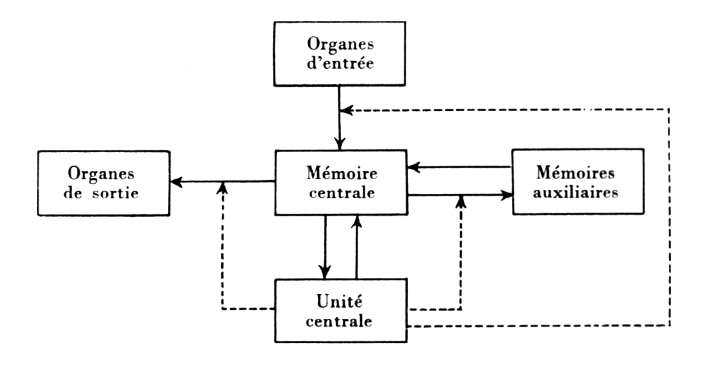
En effet, quand des informations sont fournies à l'ordinateur à partir d'un organe d'entrée, l'unité centrale « commande » la transmission de ces informations de l'organe d'entrée vers la mémoire centrale où elles seront gardées.

Quand l'ordinateur doit effectuer des calculs sur des informations gardées en mémoire centrale, l'unité centrale va « chercher » ces informations et les place dans des mémoires spéciales, appelées *registres* et liées à l'unité centrale. Les calculs s'effectuent ensuite avec les informations contenues dans ces registres et leurs résultats sont, eux aussi, gardés dans ces registres. Evidemment, ces résultats servent souvent dans des calculs ultérieurs ou seront matérialisés sur un organe de sortie. C'est pourquoi l'unité centrale peut aussi placer les contenus des registres dans la mémoire centrale.

Quand l'ordinateur a besoin d'informations se trouvant sur une mémoire auxiliaire, l'unité centrale commande la transmission de ces informations vers la mémoire centrale et, inversement, quand des informations de la mémoire centrale doivent être gardées dans une mémoire auxiliaire, la transmission se fait dans l'autre sens toujours sous le contrôle de l'unité centrale.

Enfin, quand l'ordinateur doit fournir des informations qu'il a en mémoire centrale, l'unité centrale commande la transmission de ces informations vers l'organe de sortie approprié.

Schématiquement, un ordinateur peut donc être représenté de la façon suivante :



Les flèches en trait plein symbolisent la transmission d'informations et celles en pointillé symbolisent le contrôle exercé par l'unité centrale.

I.2. Fonctionnement d'un ordinateur

L'unité centrale ne peut réaliser que des opérations très simples, telles que :

— recherche d'une information dans la mémoire centrale ;
— stockage de cette information dans un de ses registres ;
— transmission de l'information contenue dans un de ses registres vers une case bien précise de la mémoire centrale ;
— addition, soustraction, multiplication, division des contenus de deux registres ;
— comparaison des valeurs de deux registres, etc.

Or, les problèmes réels nécessitent toujours plus qu'une seule de ces opérations élémentaires. En fait, pratiquement, ils se décomposent en une succession de telles opérations élémentaires.

Exemple : La valorisation d'un stock d'articles donnés par leur quantité et leur prix (supposés gardés en mémoire centrale) se décompose, par exemple, en la succession des opérations élémentaires suivantes :

— recherche en mémoire centrale de la quantité d'un article ;
— stockage de cette quantité dans un registre A ;
— recherche en mémoire centrale du prix de cet article ;
— stockage de ce prix dans un registre B ;
— multiplication des contenus des registres A et B ;
— stockage du résultat dans un registre C ;
— transmission du contenu du registre C vers un endroit précis de la mémoire centrale ;
— comparaison du numéro de l'article examiné à celui du dernier article en stock ;
— s'il n'y a pas égalité, recommencer la première opération élémentaire avec l'article suivant, sinon s'arrêter.

Pour que l'unité centrale réalise ce travail, il faut lui donner une succession d'ordres qui auront pour effet de lui faire réaliser ces différentes opérations élémentaires.

L'ordinateur ne serait pas plus utile qu'une machine de

bureau, si ces ordres devaient être donnés à partir de l'extérieur, par exemple en appuyant sur différents boutons.

Au contraire, la première originalité d'un ordinateur est qu'il est possible d'appuyer sur un seul bouton pour démarrer un traitement, qui se poursuivra automatiquement jusqu'à la fin sans autre intervention.

Cela n'est possible que parce que l'unité centrale peut trouver quelque part les ordres qui représentent les opérations élémentaires, les comprendre et les exécuter.

Une telle succession d'ordres est appelée un *programme.* Pour éviter une structure figée, ce programme n'est pas « câblé », c'est-à-dire qu'il n'est pas réalisé par un réseau électrique immuable qui ne peut effectuer qu'un seul type de traitement. Au contraire, et c'est la seconde originalité d'un ordinateur, chaque ordre du programme est décrit par une succession de symboles (des chiffres en général) *stockés dans la mémoire centrale.* Ainsi, le programme est représenté par une succession de symboles, qui sont tous dans des cases de la mémoire centrale.

Lorsqu'on appuie sur le bouton de départ, l'unité centrale examine la succession de symboles représentant le premier ordre du programme, envoie les impulsions nécessaires dans les différents circuits pour réaliser cet ordre, puis traite de la même façon la succession de symboles représentant le second ordre et ainsi de suite jusqu'à l'exécution d'un ordre d'arrêt.

Il est très important de remarquer qu'un tel système est très souple et qu'il permet, tout en n'utilisant qu'une seule « machine », de décrire des traitements variés puisqu'une même succession d'actions élémentaires quelconques peut être stockée selon des ordres très divers et donc réaliser des traitements différents.

Comme nous l'avons dit, un ordre (on dit encore une instruction-machine) est stocké dans la mémoire centrale sous la forme d'une succession de symboles. Cette succession n'est évidemment pas quelconque ; elle obéit à des règles précises pour que l'ordinateur la déchiffre correctement et exécute l'ordre auquel elle correspond. De plus, les instructions-machine utilisables ne sont pas non plus quelconques : elles décrivent

les opérations élémentaires que l'ordinateur est capable de réaliser grâce aux circuits dont il est pourvu.

Ainsi, chaque ordinateur possède son « langage » propre, appelé *langage-machine* ; c'est un répertoire d'instructions-machine qu'il peut exécuter.

L'écriture de programmes en langage-machine est souvent très difficile. D'abord parce que ce langage est ésotérique, ensuite parce qu'il varie d'une machine à l'autre, enfin parce qu'il oblige à décomposer le problème à traiter en actions très élémentaires et qu'il en résulte une logique souvent complexe dans la succession de ces actions.

C'est pourquoi les informaticiens ont créé des langages de programmation plus proches du langage naturel mais néanmoins pas trop éloignés du langage-machine, ou du moins imaginés en tenant compte des possibilités des ordinateurs : ces langages sont appelés *langages évolués*. Le langage BASIC, objet de ce livre, en est un exemple.

Tous les langages de programmation servent à l'écriture de programmes, chargés de réaliser des travaux précis au moyen d'un ordinateur.

Ils ont sensiblement la même structure, à savoir qu'un programme est une succession d'instructions : chacune d'elles réalise une opération précise, et correspond à une certaine succession d'instructions-machine que l'ordinateur exécutera.

On retrouve aussi dans tous ces langages les mêmes types d'instructions, à savoir :

— entrées de valeurs (nombres, messages...) : voir les chapitres V, VIII et XI ;
— mémorisation de ces valeurs : voir les chapitres III, § 5 et VIII, § 3 ;
— calculs sur ces valeurs : voir les chapitres III, VIII, IX ;
— contrôle du déroulement des calculs : voir les chapitres VI, VII et X ;
— sortie de résultats : voir le chapitre IV.

Ces instructions sont, contrairement à celles du langage-machine, écrites en langage (relativement) clair.

On pourra écrire par exemple (1) une instruction de la forme :

SI QUANTITE ARTICLE = 0
ALORS IMPRIMER « STOCK EPUISE »

Il est évident qu'un programme écrit dans un langage évolué ne peut pas être compris tel quel par l'ordinateur... qui ne comprend que le langage-machine. C'est pourquoi ce programme sera traduit avant son exécution par l'ordinateur. Cette traduction est réalisée par l'ordinateur lui-même, grâce à un programme préécrit (généralement par des spécialistes, en langage-machine) qu'on appelle un *compilateur*.

Avant l'exécution de tout programme écrit en langage évolué, le compilateur est introduit en mémoire de l'ordinateur (à moins qu'il n'y figure déjà). Ce compilateur, qui est lui-même un programme, est exécuté par l'ordinateur; cela a pour effet de lire les différentes instructions du programme écrit en langage évolué et de les traduire en instructions-machine équivalentes, qui réaliseront donc la même opération mais que l'ordinateur pourra comprendre. Lorsque le programme est traduit (2), l'ordinateur peut alors l'exécuter ou plus exactement exécuter les instructions-machine qui résultent de sa traduction.

I.3. Les différentes utilisations d'un ordinateur

Un ordinateur est conçu pour pouvoir exécuter des programmes différents. Ainsi, plusieurs utilisateurs peuvent se servir du même ordinateur. Cela pose un problème de répartition dans le temps des travaux de chaque utilisateur, afin que chacun d'eux reçoive les résultats de ses programmes le plus rapidement possible.

Une première solution apportée à ce problème a reçu la dénomination anglaise de *batch processing*. Elle consiste à

(1) Ce n'est pas tout à fait le cas du langage BASIC (voir chap. VI).
(2) On dit plus volontiers que le programme a été compilé.

collecter tous les programmes écrits, par exemple pendant une demi-journée, et à les faire lire et exécuter par l'ordinateur les uns après les autres sans interruption. Cette solution est satisfaisante si tous les programmes s'exécutent normalement.

Elle présente cependant un premier inconvénient, lié aux vitesses respectives de traitement et de sortie des informations. En effet, la technologie des ordinateurs est telle que le traitement par l'unité centrale est beaucoup plus rapide que la sortie des informations sur un organe de sortie. Or, lorsqu'un programme « sort » des informations, l'unité centrale ne peut pas exécuter le programme suivant qui n'a pas été encore lu par l'ordinateur ; elle ne fait donc rien et si le nombre d'informations sorties est important, les programmes suivants sont en quelque sorte pénalisés.

Une technique permet de pallier cet inconvénient : celle de la *multiprogrammation*. Nous n'insisterons pas, cela dépassant le cadre de ce livre.

Le *batch processing* présente un second inconvénient relatif à l'état des programmes exécutés. Il serait vain de penser qu'un programme s'exécutera normalement sans erreur la première et parfois même la seconde ou troisième fois. Le plus souvent, l'ordinateur décèle lors des premières exécutions d'un programme des erreurs d'écriture ou de conception. Préalablement à l'exécution effective d'un programme, il y a donc une phase délicate de mise au point destinée à éliminer les erreurs que l'utilisateur a pu commettre. En *batch processing*, cette phase peut être très longue. En effet, lors d'un premier passage, un programme peut être précédé d'autres programmes qui prennent un temps important d'exécution. Si ce premier passage décèle uniquement des erreurs, l'utilisateur aura attendu longtemps sans obtenir les résultats qu'il souhaite. Bien plus, après avoir apporté les corrections nécessaires, il devra attendre l'exploitation suivante en *batch processing*, c'est-à-dire le plus souvent une demi-journée ; et lors de cette exploitation, le même processus peut recommencer en raison de nouvelles erreurs non décelées au premier « passage ».

Le *batch processing* est donc une technique assez mal adaptée à la mise au point de programmes.

Enfin, un autre inconvénient des solutions précédentes est qu'elles supposent que l'utilisateur dispose d'un ordinateur, suffisamment puissant pour résoudre ses problèmes et dont il supporte les charges financières.

Une technique récente a supprimé de manière satisfaisante ces inconvénients : le *temps partagé*.

Avec cette technique, plusieurs utilisateurs travaillent simultanément sur un même ordinateur.

Chaque utilisateur dispose d'une machine à écrire un peu spéciale, appelée un *terminal*.

Chaque terminal est relié à l'ordinateur au moyen d'une ligne téléphonique ordinaire, ce qui permet d'envoyer ou de recevoir des informations de l'ordinateur. Remarquons que l'ordinateur et le terminal peuvent physiquement être très éloignés l'un de l'autre.

L'ensemble constitué par l'ordinateur et les terminaux est appelé *système de temps partagé*.

Le terminal comporte le plus souvent les éléments suivants :

— un clavier de machine à écrire, avec les touches habituelles (sauf les lettres minuscules) et quelques touches spéciales dont nous verrons l'utilité dans les chapitres suivants ;
— un lecteur et un perforateur de ruban ;
— des boutons qui permettent de mettre le terminal sous tension, de faire fonctionner électriquement la machine à écrire, d'appeler l'ordinateur avec un combiné téléphonique afin de mettre l'ordinateur en liaison avec le terminal.

La mise en liaison (nous dirons plutôt la *connexion*) de l'ordinateur avec le terminal dépend le plus souvent à la fois du système de temps partagé et du type de terminal.

Lorsqu'elle a eu lieu, l'ordinateur le signale en frappant un message sur la machine à écrire et l'utilisateur peut alors commencer à se servir de l'ordinateur.

Pour cela, il doit taper sur la machine à écrire des messages qui sont envoyés à l'ordinateur ; celui-ci les examinera, exécutera ce qu'on lui demande et répondra éventuellement.

Ce dialogue est nécessaire, avant même que l'utilisateur tape un programme en langage évolué. En effet, d'une part l'utili-

sateur doit d'abord donner des renseignements qui l'identifient, puisque plusieurs utilisateurs différents peuvent se servir du même ordinateur ; d'autre part les programmes peuvent être tapés dans des langages évolués différents. L'utilisateur doit donc indiquer à l'ordinateur le langage dans lequel il désire taper un programme.

Ce dialogue est réalisé au moyen d'un langage très simple, appelé *langage de commande ou de dialogue*. Il ne faut surtout pas le confondre avec le langage BASIC : l'un, BASIC, permet d'écrire des calculs qui seront exécutés par l'ordinateur, l'autre sert à dialoguer avec l'ordinateur et en particulier à lui faire exécuter des calculs décrits par un programme écrit en langage BASIC.

Par exemple, si l'ordinateur tape sur le terminal la question :

NUMERO D'UTILISATEUR ?

l'utilisateur devra répondre en fournissant un numéro qui lui est propre.

De même, si l'ordinateur « pose » la question :

LANGAGE ?

l'utilisateur répondra en tapant, par exemple, le mot BASIC.

Les langages de dialogue diffèrent d'un système de temps partagé à un autre. Cependant, ils sont toujours constitués par un répertoire d'ordres, un ordre étant une succession de mots bien précis.

Chacun des ordres peut être tapé sur le terminal et envoyé à l'ordinateur en appuyant sur la touche « retour-chariot ».

Lorsque l'utilisateur a indiqué à l'ordinateur qu'il désirait écrire un programme dans le langage évolué dont il a donné le nom, il peut taper les différentes instructions de ce programme et les envoyer à l'ordinateur. Celui-ci examine le programme proposé, le compile *sans l'exécuter* et signale toutes les erreurs trouvées dans ce programme. Comme le programme est toujours à ce moment en mémoire de l'ordinateur, l'utilisateur peut corriger ses erreurs et proposer de nouveau un programme modifié à l'ordinateur, qui l'examinera comme ci-dessus. Ainsi,

la mise au point d'un programme en temps partagé est très facile et rapide (la compilation et les diagnostics de l'ordinateur interviennent le plus souvent dans les quelques minutes qui suivent l'entrée d'un programme) ; quand l'ordinateur ne décèle plus d'erreur dans un programme, l'utilisateur peut faire exécuter ce programme au moyen d'un ordre du langage de dialogue. Lors de cette exécution, l'ordinateur peut encore déceler des erreurs et les signaler, mais l'utilisateur peut immédiatement les corriger au moyen du langage de dialogue, car le programme se trouve toujours dans la mémoire de l'ordinateur.

Les possibilités des langages de dialogue ne se limitent pas à celles que nous avons décrites jusqu'alors.

En effet, bien souvent, un utilisateur écrit et met au point un programme qui servira plusieurs fois à des moments qui peuvent être fort éloignés dans le temps. Par exemple, la paie du personnel sera décrite par un programme qui sera exécuté tous les mois. Il serait fastidieux et long de retaper à chaque fois ce programme.

C'est pourquoi les langages de dialogue permettent le stockage de programmes quelconques.

Lorsqu'un programme « est au point », l'utilisateur peut lui donner un nom et demander à l'ordinateur de le stocker *sous ce nom* dans une mémoire auxiliaire. Lors d'une connexion ultérieure, il lui suffira de demander à l'ordinateur, au moyen d'un ordre du langage de dialogue, d'appeler le programme qu'il a stocké précédemment (en utilisant le nom qu'il lui a donné alors) et tout se passera comme si l'utilisateur avait retapé le programme.

Exemple (1) :

Première connexion
NUMERO D'UTILISATEUR ? 1327
LANGAGE ? BASIC
« écriture et mise au point d'un programme de paie »
GARDER
NOM ? PAIE
FIN

(1) Nous avons simplifié volontairement le langage de dialogue.

Lors de cette première connexion, l'utilisateur a écrit, mis au point un programme de paie et a demandé à l'ordinateur de le garder sous le nom PAIE.

Lors d'une seconde connexion (une semaine, un mois ou trois mois après, peu importe), il pourra faire exécuter ce programme par exemple avec le dialogue suivant :

```
NUMERO D'UTILISATEUR ? 1327
LANGAGE ? BASIC
CHARGER PAIE
EXECUTER
« résultats »
FIN
```

Le troisième ordre demande à l'ordinateur d'aller rechercher le programme stocké précédemment sous le nom PAIE et le quatrième de l'exécuter.

Pour terminer ce rapide aperçu sur les systèmes de temps partagé, insistons sur deux avantages qu'ils présentent.

Le premier concerne le temps de réponse de l'ordinateur.

En temps partagé, les opérations d'entrée et de sortie d'informations du terminal à l'ordinateur se font indépendamment des calculs de l'unité centrale. De plus, cette dernière « partage » son temps entre les différents utilisateurs, c'est-à-dire qu'elle accorde un certain laps de temps pour l'exécution du programme d'un premier utilisateur, puis lorsque ce laps de temps est écoulé, ou bien lorsque ce programme demande une opération d'entrée ou de sortie d'informations, elle « passe » au programme d'un second utilisateur et ainsi de suite, en examinant successivement les programmes des différents utilisateurs. Comme la vitesse de calcul de l'unité centrale est très grande par rapport aux vitesses d'opérations d'entrée ou de sortie, plusieurs utilisateurs peuvent travailler « simultanément » sans se léser les uns les autres et en ayant la sensation, à l'échelle humaine, que leurs travaux s'exécutent réellement au même instant.

Le second avantage provient de ce qu'on n'utilise qu'une partie du temps de l'ordinateur et qu'on ne paie que la partie effectivement utilisée. Ainsi, pour un même coût, l'utilisateur peut se servir d'un ordinateur plus puissant.

Chapitre II

INTRODUCTION AU LANGAGE « BASIC »

Comme introduction au langage BASIC, nous allons décrire et expliquer un programme qui résoud un problème très simple.

Nous déduirons de ce programme un certain nombre d'enseignements de base concernant la structure générale du langage Basic et des *objets* qu'il permet de manipuler.

II.1. Exemple de programme

Son but est de résoudre l'équation du second degré :

$$4x^2 - 3x - 2 = 0, \qquad (1)$$

donc de calculer et d'imprimer sur le terminal les valeurs de x qui annulent l'expression $4x^2 - 3x - 2$.

Un calcul élémentaire montre qu'il existe de telles valeurs qui sont :

$$x_1 = \frac{3 + \sqrt{41}}{8} \quad \text{et} \quad x_2 = \frac{3 - \sqrt{41}}{8}.$$

Ces deux expressions contiennent la même valeur numérique $\sqrt{41}$.

Si nous les calculions « à la main », il est bien évident que nous ne calculerions qu'une seule fois $\sqrt{41}$.

Nous allons opérer de la même façon avec l'ordinateur, en lui demandant de calculer $\sqrt{41}$ et de garder provisoirement cette valeur en vue de son utilisation dans les calculs de x_1 et de x_2.

Comme le symbole « racine carrée » n'existe pas sur le clavier du terminal, nous indiquerons à l'ordinateur qu'il doit calculer une racine carrée, en écrivant le sigle (1) SQR.

En outre, nous lui préciserons qu'il doit prendre la racine carrée du nombre 41, en faisant suivre le sigle SQR du nombre 41, *entouré de parenthèses*, c'est-à-dire SQR (41).

Cette écriture nous dispense du calcul de $\sqrt{41}$, « à la main » : l'ordinateur le fera pour nous.

Comme nous l'avons dit, il faut en outre lui demander de garder cette valeur dans sa mémoire, puisque nous l'utiliserons ultérieurement dans les calculs de x_1 et de x_2.

Pour cela, nous donnerons un *nom* à cette valeur, par exemple (2) Y, et nous demanderons à l'ordinateur de garder la valeur $\sqrt{41}$ sous le nom Y en écrivant :

Y = SQR (41)

Dans cette écriture, le signe « = » ne signifie pas, contrairement à l'habitude, que les deux quantités qui l'entourent sont égales (3), mais que la valeur numérique située à droite est le résultat d'un calcul intermédiaire, qui sera utilisé ultérieurement *sous le nom* Y.

Tout se passe comme si nous demandions à l'ordinateur :

— de réserver une « case » dans sa mémoire (peu importe où) ;
— de lui attribuer « l'étiquette » Y ;
— et d'y mettre la valeur $\sqrt{41}$.

Pour préciser clairement ce processus, nous ferons précéder la lettre Y du symbole LET.

L'écriture

LET Y = SQR (41)

représente une opération que l'ordinateur doit effectuer.

(1) Ce sigle est l'abréviation du mot anglais *Square-root.* Nous verrons dans la suite que les mots de base du langage BASIC sont tous exprimés en anglais.

(2) Le clavier du terminal ne possède pas de lettres minuscules : tous les mots employés dans le langage BASIC sont donc écrits en majuscules.

(3) Nous reviendrons sur cette difficulté qui heurte les habitudes acquises (voir chap. III).

Nous dirons que c'est une *instruction.*

Après cette première instruction, il faut demander à l'ordinateur de calculer et d'imprimer les valeurs numériques des racines x_1 et x_2.

Or, par la première instruction, l'ordinateur « sait » que la lettre Y représente la valeur $\sqrt{41}$.

Donc, au lieu de lui demander de calculer les valeurs numériques

$$\frac{3 + \sqrt{41}}{8} \quad \text{et} \quad \frac{3 - \sqrt{41}}{8}$$

il revient *maintenant* au même de lui demander de calculer les valeurs des expressions :

$$\frac{3 - \text{Y}}{8} \quad \text{et} \quad \frac{3 + \text{Y}}{8}.$$

Telles quelles, ces expressions sont peu commodes à taper sur le terminal. En outre, le langage BASIC exige que les différents symboles d'une expression soient écrits les uns à la suite des autres.

C'est pourquoi, nous remplacerons la barre de fraction par le symbole / (diviser).

De plus, nous entourerons de parenthèses les expressions 3 + Y et 3 — Y pour indiquer à l'ordinateur que leurs valeurs, résultats respectifs d'une addition et d'une soustraction, doivent être divisées par 8.

Nous écrirons donc :

(3 + Y) / 8 et (3 — Y) / 8

Pour que l'ordinateur imprime les valeurs de ces expressions, nous les ferons précéder par le mot PRINT et de plus, pour lui permettre de faire la distinction entre les deux expressions, nous les séparerons par une virgule.

L'écriture complète de l'instruction sera donc :

```
PRINT (3 + Y) / 8, (3 — Y) / 8
```

Cette instruction diffère bien sûr de l'instruction précédente :

```
LET Y = SQR (41)
```

C'est pourquoi nous écrirons ces deux instructions sur deux lignes différentes.

Enfin, nous indiquerons à l'ordinateur que son travail est terminé en écrivant le mot END sur une troisième ligne.

En résumé, les trois instructions suivantes, écrites sur trois lignes différentes, indiquent à l'ordinateur le travail qu'il doit effectuer :

```
LET Y = SQR (41)
PRINT (3 + Y) / 8, (3 — Y) / 8
END
```

Il manque encore un renseignement pour que l'ordinateur comprenne ces trois instructions. En effet, les opérations qu'elles représentent, ne peuvent pas être effectuées dans un ordre quelconque : l'ordinateur doit d'abord calculer la racine carrée de 41 et la garder sous le nom Y, avant d'imprimer les valeurs de (3 + Y) / 8 et de (3 — Y) / 8, puis s'arrêter.

Pour indiquer cette succession, nous numéroterons les trois lignes dans un ordre croissant, par exemple :

Programme 1 :

```
10  LET Y = SQR (41)
20  PRINT (3 + Y) / 8, (3 — Y) / 8
30  END
```

Cette écriture représente un programme, qui calculera et imprimera les racines de l'équation $4x^2 - 3x - 2 = 0$.

Nous pouvons en déduire la structure générale d'un programme : c'est une succession de lignes ; chacune d'elles est constituée d'un numéro suivi d'une instruction.

Les numéros sont des nombres entiers de cinq chiffres au plus, compris entre 1 et 99999. Ils donnent l'ordre dans lequel l'ordinateur exécutera les instructions : il commencera par la ligne de numéro le plus bas, puis il effectuera l'instruction de

la ligne de numéro immédiatement supérieur et ainsi de suite (1).

Les numéros ne sont pas nécessairement consécutifs : leur choix ne détermine qu'un ordre de déroulement des instructions du programme (voir le Programme 1).

Notons que, lors de l'entrée du programme sur le terminal, les lignes peuvent être tapées dans un ordre quelconque : l'ordinateur classe automatiquement toutes les lignes entrées par numéros croissants.

Par exemple, le Programme 1 peut être introduit sous la forme suivante :

```
20  PRINT (3 + Y) / 8, (3 — Y) / 8
30  END
10  LET Y = SQR (41)
```

Cette remarque est très importante : elle permet d'intercaler à tout moment de nouvelles lignes entre deux lignes préalablement introduites dans l'ordinateur.

Ainsi, dans l'exemple précédent, après la frappe de la ligne 10, nous pouvons encore intercaler une ligne entre les lignes 20 et 30. Il suffit de donner à cette nouvelle ligne un numéro compris entre 20 et 30.

Dans les chapitres suivants, nous décrirons les différentes instructions du langage BASIC. Cette description sera, d'une part *syntaxique*, c'est-à-dire qu'elle indiquera la façon d'écrire correctement chaque instruction, d'autre part *sémantique*, c'est-à-dire qu'elle expliquera ce que fait effectivement l'ordinateur lorsqu'il exécute telle ou telle instruction.

Donnons encore quelques remarques sur l'écriture d'un programme. Lors de la frappe du programme, chaque ligne doit être envoyée à l'ordinateur. Ceci est réalisé en terminant la ligne par la frappe de la touche « retour-chariot » (marquée RETURN sur le clavier). Son effet est de repositionner la tête d'écriture du terminal au début de la ligne suivante.

Il n'est pas possible de prolonger la frappe d'une instruction sur plusieurs lignes. Par conséquent, toute ligne d'un programme, y compris le caractère « retour-chariot » doit contenir

(1) Certaines instructions peuvent modifier cet ordre (voir chap. VI, VII et X).

au plus soixante-douze caractères, nombre maximal de caractères d'une ligne du terminal.

Dans l'écriture (et la frappe) d'une instruction, les « espaces » obtenus en frappant sur la barre d'espacement, sont ignorés par l'ordinateur, sauf dans un cas très particulier (voir chap. IV).

Ainsi, les deux instructions suivantes sont strictement équivalentes pour l'ordinateur (1) :

```
LETY=SQR(41)
```

et

```
LET ⊔ Y = ⊔ S ⊔ Q ⊔ R ⊔ (4 ⊔ 1)
```

Il est possible d'insérer des commentaires dans un programme, pour en faciliter la compréhension.

Une ligne de commentaires est constituée d'un numéro suivi du symbole REM et d'un texte quelconque permettant au programmeur de décrire succinctement ce qu'une partie du programme est censée faire.

Exemple :

```
 5  REM RESOLUTION D'UNE EQUATION DU SECOND
 7  REM DEGRE
10  LET Y = SQR (41)
15  REM IMPRESSION DES RACINES
20  PRINT (3 + Y) / 8, (3 — Y) / 8
30  END
```

Les lignes de commentaire sont complètement ignorées par l'ordinateur. En particulier, elles peuvent être insérées à n'importe quel endroit d'un programme.

II.2. Les objets manipulés en BASIC

Les instructions décrivent des opérations simples que l'ordinateur va effectuer sur certains « objets ».

Le Programme 1 fournit deux types d'objets que nous allons

(1) Pour plus de clarté, nous notons l'espace par ⊔.

maintenant décrire de manière plus précise. Ce ne sont pas les seuls que le langage BASIC permet de manipuler : nous en décrirons d'autres plus complexes aux chapitres III, VIII et IX.

Un premier type d'objets que nous rencontrons dans le Programme 1 est celui représenté par la lettre Y.

Nous avons déjà dit que cette lettre désignait le « nom » d'une case dans la mémoire de l'ordinateur et que l'ordinateur y placerait une valeur numérique.

Un tel objet est appelé une *variable simple.*

Son utilisation est fondamentale. En effet, certaines valeurs utilisées dans un programme peuvent ne pas être connues *a priori*, soit parce que leur calcul à la main est fastidieux et inutile, par exemple $\sqrt{41}$ dans le Programme 1, soit parce qu'elles peuvent varier au cours des calculs (voir les chap. III, V, VIII...).

Il est donc nécessaire pour pouvoir effectuer des calculs sur de telles valeurs, de les représenter par des symboles abstraits.

Ainsi, lorsque nous écrivons Y = SQR (41), nous ignorons la valeur exacte de $\sqrt{41}$, mais nous pouvons l'utiliser implicitement dans les calculs ultérieurs grâce à la variable simple Y (cas de la ligne 20).

Une variable simple est donc un symbole qui désigne une valeur numérique, ou plus précisément, qui *repère* dans la mémoire de l'ordinateur une « case » *contenant* une valeur numérique.

Le langage BASIC permet l'emploi de plusieurs variables simples dans un même programme.

Leur règle d'écriture est la suivante :
le nom d'une variable simple peut être :

— soit une lettre seule ;
— soit une lettre suivie d'un chiffre.

Ainsi, A, B 1 ou X 2 sont des variables simples correctes, alors que DELTA ou 3B n'en sont pas.

Deux variables simples différentes repèrent deux cases différentes dans la mémoire de l'ordinateur.

Nous trouvons dans le Programme 1 un second type

d'objets : les nombres explicitement écrits tels que 41 ou 8.

Cette possibilité est aussi fondamentale puisque l'ordinateur effectue des calculs sur des nombres.

En BASIC, il est possible d'écrire explicitement deux types de nombre : les nombres entiers et les nombres réels.

Un nombre entier est une succession de chiffres dont le premier est différent de 0.

Exemple : 2437.

Un nombre réel peut s'écrire sous deux formes :

i) *La forme décimale.* — Elle est constituée d'une partie entière (nombre entier), puis d'un point et enfin d'une partie décimale (nombre entier).

Exemple : le nombre 16,3748 s'écrira dans un programme : 16.3748.

ii) *La forme « virgule flottante ».* — Le nombre 16,3748 peut encore s'écrire de manière équivalente :

$1{,}63748 \times 10$, ou $0{,}163748 \times 10^2$...

c'est-à-dire au moyen du produit d'un nombre sous forme décimale par une puissance de 10.

Par rapport à l'écriture 16,3748, la virgule a été déplacée (a « flotté »), mais la puissance de 10 permet de rétablir la même valeur pour le nombre. De même, le nombre 16,3748 peut aussi s'écrire :

$\frac{163{,}748}{10}$ c'est-à-dire avec un symbolisme classique :

$$163{,}748 \times 10^{-1}$$

ou encore $\frac{1637{,}48}{100}$ c'est-à-dire $1637{,}48 \times 10^{-2}$...

Toutes ces écritures ont la même forme, appelée virgule flottante, à savoir :

un nombre entier ou sous forme décimale, suivi d'une puissance de 10, positive ou négative.

Leur écriture en BASIC est très analogue. Elle est constituée :

— d'un nombre entier ou sous forme décimale ;
— puis d'un symbole E représentant 10 ;
— puis d'un signe + ou — ;
— enfin d'un exposant (nombre entier).

Exemples : le nombre 16,3748 pourra s'écrire en virgule flottante :

1.63748 E + 1 ou 0.163748 E + 2...
163.748 E — 1 ou 1637.48 E — 2...

Remarquons que le signe « + » de l'exposant peut être omis. L'écriture 0.163748 E + 2 est équivalente à 0.163748 E 2.

Il est possible d'écrire des nombres ayant autant de chiffres que l'on veut. Cependant, s'il y en a trop, l'ordinateur ne pourra pas tous les garder en mémoire : le nombre maximal de chiffres que l'ordinateur peut garder en mémoire pour décrire un nombre quelconque, dépend du système de temps partagé.

Lorsque la partie décimale d'un nombre réel contient trop de chiffres, l'ordinateur ignore les chiffres excédentaires.

Exemple : si l'ordinateur ne peut conserver que huit chiffres par nombre, l'écriture 12.3456789 dans un programme correspondra en fait au nombre 12.345678 gardé dans la mémoire.

Lorsqu'un nombre entier contient trop de chiffres, l'ordinateur le transforme d'abord sous forme virgule flottante en ne prenant qu'un seul chiffre avant la virgule, puis il ignore dans le nombre transformé les chiffres excédentaires.

Exemple : le nombre 123456789 écrit dans un programme sera gardé en mémoire sous la même forme que s'il avait été écrit :

1.2345678E8

Nous n'avons parlé dans ce qui précède que des nombres positifs.

Il est possible aussi d'écrire et d'utiliser des nombres négatifs.

Il suffit pour cela de faire précéder du signe « — » l'écriture du nombre telle qu'elle a été décrite ci-dessus.

Exemples :

— 163
— 1.6348
— 163.48 E — 2

EXERCICES

— Qu'est-ce qu'une variable simple ? Quel est son intérêt ? Quelle est sa règle d'écriture en BASIC ?

— Comment repère-t-on les lignes d'un programme ?

— Peut-on introduire en machine un nombre aussi grand que l'on veut ?

— Si, dans un programme, vous avez à utiliser plusieurs fois la même expression :
  — la réécrivez-vous à chaque fois ?
  — avez-vous une autre attitude ? laquelle ? exemple.

— Quels sont les deux types d'instruction rencontrés dans ce chapitre ? A quoi servent-elles ?

— Intérêt de la numérotation des lignes d'un programme ?

— Qu'est-ce qu'un programme « correct » en langage BASIC ?

— Comment peut-on introduire un commentaire ?

— Dans les types d'instruction décrits dans ce chapitre, quelle est la signification des blancs ?

Chapitre III

LES CALCULS EN « BASIC ». INSTRUCTION D'AFFECTATION

Nous avons décrit au chapitre précédent deux types d'objets manipulés en BASIC. En fait, ils représentent tous deux des valeurs numériques sur lesquelles il est possible d'effectuer des opérations.

Dans le présent chapitre, nous décrirons les calculs que l'on peut effectuer sur ces valeurs numériques, puis un nouveau type d'objet représentant aussi une valeur numérique. Enfin, nous reviendrons sur l'instruction qui permet de désigner une valeur numérique par une variable.

III.1. Les expressions arithmétiques

Elles permettent de décrire des opérations à effectuer entre des nombres ou des variables (1) ou d'autres entités que nous verrons au § 2 : tous ces éléments représentant des valeurs numériques sont appelés des *opérandes*.

En BASIC, cinq opérations sont permises. Elles sont représentées par cinq symboles appelés *opérateurs* :

+ pour l'addition de deux opérandes (2 + 3) ;
— pour la soustraction de deux opérandes (2 — 3) ;
* pour la multiplication de deux opérandes (2 * 3) ;
/ pour la division de deux opérandes (2 / 3) ;
↑ pour l'élévation d'un opérande à la puissance d'un autre opérande (2 ↑ 3 représente l'écriture courante 2^3).

(1) Ce terme est pris en un sens plus général que celui de variable simple. Il englobe un autre type de variable que nous verrons au chapitre IX.

Chaque opérateur symbolise donc une opération entre *deux* opérandes, par exemple X + 3 ou X * Y...

La valeur numérique du résultat est obtenue en effectuant l'opération sur les valeurs numériques des deux opérandes. Si un opérande est un nombre explicitement écrit, sa valeur numérique est immédiatement connue. Si c'est une variable simple, sa valeur numérique est celle repérée par cette variable simple, *au moment où l'opération est effectuée* (1).

III.2. Ordre des opérations dans une expression

Le résultat d'une opération étant une valeur numérique, il peut, à son tour, être un opérande pour une autre opération, c'est-à-dire qu'il est possible d'effectuer plusieurs opérations successives dans une même expression.

Exemples :

X + 3 + Y
X + 3 * Y

Cette possibilité entraîne une difficulté quant à la compréhension de l'expression et de l'ordre dans lequel les différentes opérations doivent être effectuées.

Dans le premier exemple ci-dessus, cet ordre est indifférent. En effet, le résultat est le même que l'on effectue d'abord l'addition de X et de 3, et que l'on ajoute le résultat obtenu à Y, ou bien que l'on effectue d'abord la seconde addition de 3 et de Y et que l'on ajoute le résultat obtenu à X.

Par contre, dans le second exemple, on n'obtiendra pas le même résultat si l'on commence par la multiplication ou par l'addition. Il suffit, pour s'en convaincre, de donner des valeurs numériques à X et Y : 2 à X et 3 à Y. Si la première opération effectuée est l'addition, on obtient 15 comme résultat. Au contraire, si la première opération effectuée est la multiplication, on obtient 11.

(1) Cf. § III.5.

Il est donc essentiel d'adopter des conventions qui permettront à l'ordinateur d'effectuer les calculs sans aucune ambiguïté.

Ces conventions consistent à définir l'ordre des opérations en attribuant une priorité à chaque opérateur. Intuitivement, si un opérateur a une priorité plus grande qu'un autre, l'opération qu'il représente sera effectuée la première.

Plus précisément, l'ordinateur cherche quelle est la première opération à effectuer dans l'expression, de la manière suivante : il examine l'expression de la gauche vers la droite et considère le premier opérateur rencontré.

Si celui-ci n'est suivi d'aucun opérateur, ou bien s'il a une priorité supérieure ou égale à celle de l'opérateur qui le suit, il représente la première opération à effectuer.

En cas contraire, donc si le premier opérateur rencontré a une priorité inférieure à celle de l'opérateur qui le suit, l'ordinateur abandonne provisoirement le premier opérateur rencontré et recommence le processus décrit ci-dessus avec les autres opérateurs de l'expression.

Exemple : En supposant que $\uparrow$ a une priorité plus grande que celle de $\times$, elle-même plus grande que celle de $+$, la première opération à effectuer dans l'expression $X + 3 \times Y \uparrow 2$ sera $\uparrow$.

En effet, le premier opérateur rencontré par le calculateur dans son examen est $+$. Comme sa priorité est inférieure à celle de l'opérateur $\times$ qui le suit immédiatement, $+$ est provisoirement abandonné et $\times$ devient donc le premier opérateur rencontré. Sa priorité étant inférieure à celle de l'opérateur $\uparrow$ qui le suit, $\times$ est lui aussi provisoirement abandonné et $\uparrow$ devient le premier opérateur rencontré. Comme il n'est suivi d'aucun opérateur, il représente bien la première opération à effectuer.

La première opération à effectuer étant trouvée, l'ordinateur l'effectue et remplace l'écriture de cette opération par une variable *fictive* qui symbolise le résultat de l'opération. Cette variable n'a rien à voir avec l'écriture de l'expression dans un programme. Nous l'introduisons artificiellement pour faciliter la compréhension du calcul d'une expression.

Dans l'exemple ci-dessus, après le calcul de Y ↑ 2 dont nous symbolisons le résultat par R1, l'expression X + 3 * Y ↑ 2 sera remplacée fictivement par l'expression X + 3 * R1.

Ce processus a donc pour effet de diminuer de un le nombre d'opérateurs dans l'expression. Il suffit d'appliquer le même processus sur la nouvelle expression pour trouver la seconde opération à effectuer, et ainsi de suite jusqu'à ne plus obtenir, après les réductions successives, qu'une expression ne contenant qu'un opérateur. Cet opérateur représente la dernière opération à effectuer dont le résultat est celui de l'expression.

Reprenons l'exemple précédent :

La seconde opération à effectuer est * puisque dans l'expression X + 3 * R1, * a une priorité plus grande que +.

En symbolisant par R2 le résultat de 3 * R1, l'expression devient X + R2; comme il ne reste plus que l'opérateur +, le calcul sera terminé en additionnant les valeurs de X et de R2.

Les processus précédents se schématisent par :

R1	résultat de	Y ↑ 2
R2	résultat de	3 * R1
R3	résultat de	X + R2
R3	résultat final de l'expression	X + 3 * Y ↑ 2

Dans le langage BASIC, les priorités des opérateurs sont les suivantes : + et — ont même priorité ; cette priorité est inférieure à celle de * et de / qui ont entre eux même priorité ; cette dernière est elle-même inférieure à celle de ↑.

Ce choix n'est pas arbitraire et correspond à la compréhension courante des expressions.

Signalons un cas particulier qui conduit souvent à une erreur d'interprétation.

L'écriture en BASIC de l'expression X + Y / 4 * Z correspond à l'écriture courante $X + \frac{YZ}{4}$ et non à $X + \frac{Y}{4Z}$.

Une écriture BASIC de cette dernière expression pourrait être :

X + Y / 4 / Z

Remarquons aussi que tous les opérateurs doivent être

explicitement écrits. Par exemple, l'ordinateur ne comprendra pas l'expression :

$$X + Y / 4Z$$

puisque, pour lui, 4Z devrait être un opérande : or, ce n'est ni un nombre, ni une variable (qui doit commencer par une lettre).

III.3. Parenthèsage des expressions

Les exemples d'écriture d'une expression rencontrés ci-dessus heurtent parfois les habitudes ; c'est le cas de $X + Y / 4 / Z$. Il peut être préférable d'écrire que Y est divisé par le résultat du produit $4 \ast Z$, c'est-à-dire que ce résultat est un opérande de la division.

Cela revient à imposer que le résultat d'une ou de plusieurs opérations soit un opérande d'une autre opération, indépendamment des priorités respectives des différents opérateurs.

Cette possibilité existe en langage BASIC : il suffit d'entourer de parenthèses la partie de l'expression que l'on veut considérer comme un opérande (1).

Par exemple :

$X + \frac{Y}{4Z}$ sera écrit $X + Y / (4 \ast Z)$.

Bien entendu, cette possibilité n'est pas limitée à une seule partie de l'expression. Elle peut s'appliquer à plusieurs parties d'expression, que nous appellerons aussi des sous-expressions.

Exemple :

$(4 \ast (X + Y / 3) + Z) / (8 + X)$

L'existence de parenthèses dans une expression modifie l'ordre dans lequel les opérations seront effectuées.

L'ordinateur opère comme suit :

Il cherche la première parenthèse fermante en examinant

(1) Nous avons déjà rencontré cette possibilité à la ligne 30 du Programme 1 (cf. chap. II).

l'expression de la gauche vers la droite. L'ayant trouvée, il revient en arrière et cherche la première parenthèse ouvrante. Ces deux parenthèses encadrent une sous-expression sans parenthèses dont la valeur est calculée au moyen du processus décrit précédemment.

L'ordinateur remplace alors *fictivement* toute la sous-expression, y compris les parenthèses qui l'encadrent, par une seule variable qui symbolise le résultat du calcul. Il obtient ainsi une nouvelle expression contenant deux parenthèses de moins que l'expression de départ et il recommence le processus sur la nouvelle expression.

Exemple : considérons l'expression :

(4 × (X + Y / 3) + Z) / (8 + X)

L'ordinateur isole d'abord la sous-expression X + Y / 3.

Il en calcule la « valeur » par le processus décrit plus haut.

En symbolisant cette valeur par la variable R1, il remplace l'expression de départ par :

(4 × R1 + Z) / (8 + X)

Dans celle-ci, l'ordinateur isole la sous-expression 4 × R1 + Z, dont il calcule la valeur et ainsi de suite.

Nous n'avons pas parlé jusqu'alors d'un cas particulier d'expressions, dont l'importance apparaîtra au § 5 et aux chapitres suivants : c'est le cas où l'expression est réduite à un seul opérande. Sa valeur est alors obtenue immédiatement sans aucune opération.

En résumé, une expression arithmétique est constituée :

— soit d'un seul opérande,
— soit d'une succession d'opérandes, d'opérateurs et de parenthèses, vérifiant les trois règles suivantes :
  règle 1 : L'expression ne commence pas par un opérateur ;
  règle 2 : L'expression ne contient pas deux opérateurs consécutifs ;
  règle 3 : L'expression contient autant de parenthèses fermantes que de parenthèses ouvrantes.

La règle 1 souffre une seule exception : celle de l'opérateur —. Il est en effet permis de commencer une expression par l'opérateur — suivi d'un opérande, ou d'une expression entre parenthèses (1). Dans ce cas, l'opérateur — sert à donner la valeur opposée de la valeur de l'opérande ou du résultat de l'expression entre parenthèses.

Exemples :

— X,
— (X + 3 × Y).

III.4. Les appels de fonctions

Nous avons dit au § 1 que les opérandes pouvaient être des nombres, des variables ou d'autres entités que nous allons décrire maintenant.

Nous avons déjà vu un exemple de telles entités à la ligne 10 du Programme 1 (cf. chap. II). Dans cet exemple est écrit : SQR (41) ; on dit que c'est un appel de *fonction standard.*

Plus généralement, une fonction standard est une fonction mathématique classique, représentée par un sigle, et dont l'ordinateur calcule automatiquement la valeur quand on fournit une valeur numérique, à la variable de la fonction.

Ainsi, quand nous écrivons SQR (41), l'ordinateur calcule la valeur de la fonction standard « racine carrée », de sigle SQR, pour la valeur 41 de la variable. Cette valeur est appelée *paramètre* de la fonction.

Un appel de fonction standard fournit donc une valeur numérique et, par conséquent, peut constituer un opérande d'une opération. La forme la plus générale est la suivante :

sigle de la fonction standard, suivi du paramètre entre parenthèses. Ce paramètre peut être une expression quelconque.

(1) Cette utilisation de l'opérateur — est en fait plus large que celle que nous donnons. Nous la restreignons dans un but de clarté.

En BASIC, il est possible d'utiliser les fonctions standards suivantes :

— la fonction racine carrée, de sigle SQR ;
— les fonctions trigonométriques sinus, cosinus et tangente de sigles respectifs (1) SIN, CØS et TAN.

L'ordinateur considère que la valeur du paramètre fourni est celle d'un arc *exprimé en radians*. Ainsi la valeur de SIN (3.1415926) est égale à 0 (sinus du nombre π). Par contre, SIN (180) est non nul ;
— la fonction arctangente, de sigle ATN, donne la valeur en radians, comprise entre $-\frac{\pi}{2}$ et $+\frac{\pi}{2}$, de l'arc dont la tangente est égale à la valeur du paramètre. Ainsi ATN (1) est égal à 0.785398, c'est-à-dire $\frac{\pi}{4}$;
— l'exponentielle en base *e*, de sigle EXP ;
— les logarithmes népérien et décimal de sigles respectifs LØG et LGT (2) ;
— la valeur absolue d'un nombre, de sigle ABS.

Par exemple, ABS (— 2) est égal à 2.

Cette fonction standard est utile car bien souvent le signe de la valeur d'une expression contenant des variables n'est pas connu *a priori*. Par exemple, on peut savoir si la valeur d'une variable X est positive en testant si elle est égale ou non à ABS (X) (cf. chap. VII) ;
— la fonction, de sigle INT, donnant la valeur du plus grand nombre entier inférieur ou égal à celle du paramètre. Par exemple, INT (4.8) vaut 4. La fonction INT permet de calculer l'arrondi d'un nombre réel, que nous représenterons par exemple par la variable X (ce peut être une expression quelconque), c'est-à-dire la valeur entière I la plus proche de X. Si la valeur de X est positive, son arrondi est représenté par l'écriture INT (X + 0.5). En effet, si la valeur de X est comprise entre I — 0.5 et I, alors X + 0.5 est

(1) Pour distinguer le chiffre 0 de la lettre O, nous écrirons cette dernière Ø.
(2) Cette fonction standard n'existe pas sur tous les systèmes de temps partagé.

compris entre I et I + 0.5. Donc, INT (X + 0.5) vaut bien I. Si, au contraire, la valeur de X est comprise entre I et I + 0.5, alors X + 0.5 est compris entre I + 0.5 et I + 1 et on obtient encore le bon résultat. Il faut cependant remarquer que la méthode précédente arrondit les valeurs équidistantes de deux entiers, par exemple 1.5 à la valeur entière *supérieure*.

Un raisonnement analogue montre que si la valeur de X est négative, son arrondi est représenté par l'écriture INT (X — 0.5) ;

— la fonction « signe » (1), de sigle SGN, qui donne le signe du paramètre. Elle vaut :

0 si la valeur du paramètre est nulle ;
— 1 si la valeur du paramètre est négative ;
+ 1 si la valeur du paramètre est positive.

En combinant les fonctions SGN, INT et ABS, on obtient l'arrondi d'un nombre réel X par la seule écriture :

SGN(X) ⋆ INT(ABS(X) + 0.5).

Rappelons que le paramètre d'un appel de fonction standard peut être une expression, donc en particulier contenir des appels de fonctions standards.

Exemple :

SQR(1 + CØS(2 ⋆ X) ↑ 2)

L'intérêt des fonctions standards est de pouvoir calculer facilement les valeurs d'une fonction. Cependant leur liste est limitée aux quelques fonctions classiques indiquées plus haut.

Il peut arriver aussi qu'il faille calculer plusieurs fois dans un programme une même expression, et qu'entre ces différents calculs, seule la valeur d'une variable ait changé.

Exemple : Il faut calculer les valeurs de l'expression (EXP(X) + 3 ⋆ Y) / Z pour plusieurs valeurs de X, par exemple 1, 2.46, — 3...

(1) Cette fonction standard n'existe pas sur tous les systèmes de temps partagé.

Au lieu de recopier à chaque fois cette expression, il est préférable d'indiquer au calculateur que nous créons une nouvelle fonction, dont la variable est X et dont l'expression est celle donnée ci-dessus.

Pour pouvoir utiliser cette fonction dans le programme, il faut lui donner un nom : il est constitué obligatoirement des lettres FN (qui rappelle que le nom est celui d'une fonction) suivies d'une lettre quelconque, par exemple FNA. Ceci permet donc d'utiliser 26 fonctions différentes dans un même programme.

La création d'une fonction se fait au moyen d'une instruction dont le schéma est le suivant :

DEF nom de la fonction (lettre) = expression.

Exemple :

```
90  DEF FNA(X) = (EXP(X) + 3 * Y) / Z
```

La lettre entre parenthèses désigne le « paramètre » de la fonction, qui doit bien sûr apparaître dans l'expression à droite du signe =.

Cette instruction doit être précédée d'un numéro de ligne et doit pouvoir être tapée toute entière sur une seule ligne.

Elle peut cependant être placée à un endroit quelconque du programme. En effet, elle n'entraîne aucune action spécifique de la part de l'ordinateur qui ne fait qu'enregistrer le nom et l'expression de la fonction.

Lorsqu'une fonction a été ainsi créée dans un programme, elle peut être utilisée comme opérande dans une expression, au moyen d'un appel de cette fonction. Cet appel est analogue à celui d'une fonction standard : il est constitué du nom de la fonction suivi, entre parenthèses, d'une valeur du paramètre.

Exemple :

```
 90  DEF FNA(X) = (EXP(X) + 3 * Y) / Z
120  V = (3 + FNA(1) * V) / (2 * P + 5)
```

Dans cette écriture, FNA (1) est un opérande dont la valeur est celle de l'expression à droite du signe = dans la définition

de la fonction FNA, lorsqu'on donne à X la valeur 1. (Cette valeur peut être bien sûr celle d'une expression plus compliquée.)

S'il apparaît dans l'expression qui définit la fonction, d'autres variables (ici Y et Z), leurs valeurs dans le calcul sont alors celles qu'elles ont au moment de l'appel de la fonction.

Enfin, remarquons que dans l'expression de définition d'une fonction, peuvent apparaître d'autres fonctions définies par ailleurs.

Cependant, une fonction ne peut pas « s'appeler elle-même » directement comme dans :

```
10   DEF FNA(X) = (FNA(X) + 3 * Y) / Z,
```

ou indirectement comme dans :

```
10   DEF FNA(X) = (FNB(Z) + 3 * Y) / Z
20   DEF FNB(Z) = (FNA(2 * X) + Y) * (X + 3).
```

III.5. L'instruction d'affectation

Revenons sur l'instruction figurant à la ligne 10 du Programme 1. Son intérêt est de donner un nom au résultat du calcul de SQR (41).

Nous dirons que la valeur numérique du second membre à droite du signe =, est affectée à la variable figurant à gauche de ce signe.

Une telle instruction est appelée instruction d'affectation. Sa forme la plus générale est (1) :

```
LET   variable = expression.
```

Elle signifie, qu'après l'exécution de cette instruction, la valeur numérique repérée par la variable à gauche du signe = est celle de l'expression à droite de ce signe, et ceci jusqu'à nouvelle instruction.

Une instruction d'affectation change donc la valeur repérée par une variable. Dans un programme, il peut bien sûr y avoir

(1) Le terme de variable est pris ici dans un sens très général. Dans certains systèmes de temps partagé, il est possible d'omettre le mot LET.

plusieurs instructions d'affectation, où la même variable figure à gauche du signe =.

Exemple : Considérons le programme suivant :

```
10  LET X = 1
20  PRINT X
30  LET X = 2
40  PRINT X
50  END
```

Au cours de l'exécution de ce programme, l'ordinateur imprime *d'abord* la valeur 1 affectée à X à la ligne 10, *puis* la valeur 2 affectée à X ultérieurement, à la ligne 30, avant l'instruction d'impression de la ligne 40.

Lorsque l'ordinateur exécute une instruction d'affectation, il ne se préoccupe pas d'abord de la variable à gauche du signe =. Il calcule la valeur de l'expression à droite du signe =, puis place cette valeur dans la « case » représentée par la variable du premier membre.

Ainsi, l'instruction suivante n'est pas incohérente :

```
LET X = X + 1.
```

Son effet est d'abord de calculer la valeur de X + 1, c'est-à-dire d'ajouter 1 à la valeur repérée par X *avant* l'exécution de cette instruction, puis de mettre le résultat obtenu dans la « case » représentée par X.

Ainsi, si la valeur de X *avant* cette instruction est 2, sa valeur *après* sera 3.

La variable X joue ici le rôle d'un compteur dans lequel on cumule la valeur 1. Nous reviendrons plus loin (cf. chap. VII) sur ce type d'utilisation.

Faisons une dernière remarque sur les instructions d'affectation : lors de l'exécution d'une telle instruction, il est nécessaire que l'expression à droite du signe = soit calculable, c'est-à-dire que toutes les variables qui y figurent aient reçu *préalablement* une valeur, soit par affectation, soit par d'autres moyens que nous examinerons plus loin (cf. chap. V et VIII).

Si une telle variable n'a pas reçu préalablement de valeur, l'ordinateur le signale par un message, ou bien affecte automatiquement la valeur 0 à cette variable.

EXERCICES

— Quelle est la valeur de l'expression écrite en BASIC :
— 4 ⋇ A ⋇ C ⋇ B1 / (2 ⋇ A) ⋇ (B2 ↑ 2 — A ⋇ B1 / B2)
pour A = 10, C = 2, B1 = 20 et B2 = 10 ?

— Ecrire les formes BASIC des expressions suivantes :

$$\frac{\frac{A}{B}}{\frac{C}{D}}, \qquad \frac{ax + b}{cx + d}, \qquad \frac{\frac{(9x^2 - 4)}{5} + t}{\frac{(9x^2 - 4)}{t} + 5}.$$

— Dans la dernière expression de l'exercice ci-dessus, on suppose t fixé. Mettre cette expression sous forme d'une fonction utilisable ultérieurement dans une expression arithmétique. Imaginer une petite séquence d'utilisation.

Mêmes questions pour l'expression $\frac{ax + b}{cx + d}$.

— Quel est l'intérêt d'une instruction d'affectation ? Quel est son effet en machine ?

Chapitre IV

SORTIES DES RÉSULTATS. INSTRUCTION « PRINT »

Un type d'instructions essentiel dans un langage de programmation est celui qui réalise l'impression des résultats d'un calcul.

En BASIC, cela se fait au moyen de l'instruction PRINT, que nous avons déjà utilisée à la ligne 20 du Programme 1.

Dans le présent chapitre, nous étudions plus en détail les différentes formes de cette instruction.

IV.1. Impression d'une ou de plusieurs valeurs

La forme la plus simple de l'instruction PRINT consiste en l'écriture du seul mot PRINT.

Exemple :

```
10  PRINT
```

Son effet est de faire avancer d'une ligne le papier du terminal et de replacer la tête d'écriture en début de ligne.

L'écriture successive d'instructions de ce type fait avancer le papier du terminal d'autant de lignes qu'il y a d'instructions PRINT.

Hormis ce cas particulier, les autres formes de l'instruction PRINT comportent au moins un symbole après le mot PRINT.

Parmi ces autres formes, la plus simple est celle qui permet d'imprimer une seule valeur. Schématiquement, la représentation est :

PRINT expression

Exemple :

```
10 PRINT (4 * X ↑ 2 + 3 * X) / Z
```

L'exécution de cette instruction se fait en trois phases. L'ordinateur :

— calcule la valeur de l'expression ;
— *imprime le nombre obtenu à partir de l'endroit où la tête d'écriture est positionnée à ce moment-là ;*
— *ramène cette tête d'écriture au début de la ligne suivante.*

La forme sous laquelle les nombres sont imprimés sur le terminal dépend du système de temps partagé.

En règle générale, un nombre entier comportant trop de chiffres pour être imprimé tel quel, sera imprimé sous la forme virgule flottante (cf. § II.2).

Un nombre réel pourra aussi, suivant sa valeur, être imprimé sous la forme virgule flottante.

Exemple : le programme suivant

```
10 LET I = 123456789
20 PRINT I
30 PRINT I / 2
40 END
```

imprimera par exemple :

```
.12345678 E + 08
.61728945 E + 07
```

D'autre part, si les chiffres précédant le point décimal sont nuls, ils ne sont pas imprimés.

Exemple :

le nombre 0.356 sera imprimé .356
le nombre — 0.356 sera imprimé — .356.

Il est également possible de faire imprimer plusieurs nombres sur une même ligne.

La forme de l'instruction correspondante est la suivante :

PRINT liste d'expressions

Une liste d'expressions est une succession d'expressions *séparées par des virgules.*

Exemple :

10 PRINT 2 * X, (3 * Y + 4) / Z, EXP (X) + Y

Pour comprendre l'effet de cette instruction, remarquons d'abord que chaque ligne du terminal est divisée en cinq zones de quinze caractères.

Lors de l'exécution de l'instruction PRINT, l'ordinateur :

— calcule la valeur de la première expression et l'imprime dans la première zone (1) ;
— ramène la tête d'écriture au début de la ligne suivante, si l'instruction est terminée ;
— positionne la tête de lecture, au début de la zone suivante, si la première expression est suivie d'une virgule.

Ceci fait, il calcule la valeur de la seconde expression et l'imprime à partir de l'endroit où se trouve à ce moment-là la tête de lecture (c'est-à-dire dans la seconde zone), et ainsi de suite jusqu'à épuisement de la liste d'expressions.

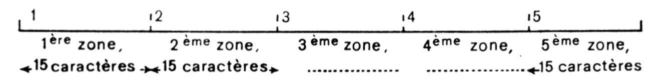
Si cette liste comporte plus de cinq expressions, donc plus de cinq valeurs à imprimer, le calculateur ramène automatiquement la tête d'écriture au début de la ligne suivante pour l'impression de la sixième valeur ; et l'impression des valeurs suivantes se fera selon le même processus.

Exemples :

1) En admettant que les valeurs des variables X1, X2, X3, X4 et X5 sont respectivement 1, 2, 3, 4, 5, l'effet de l'instruction :

   10 PRINT X1, X2, X3, X4, X5

   est d'imprimer :



(1) Nous supposons que la tête d'écriture est positionnée au début d'une ligne.

2) Si, de plus, les valeurs des variables X6 et X7 sont respectivement 6 et 7, l'effet de l'instruction

   10 PRINT X1, X2, X3, X4, X5, X6, X7

   est de remettre la tête d'écriture au début de la ligne suivante après avoir imprimé les deux lignes :

```
|1          |2          |3          |4          |5          |
|6          |7
```

Il est possible de ne pas ramener la tête d'écriture au début de la ligne suivante après l'exécution d'une instruction PRINT. Nous en verrons l'intérêt au chapitre VIII.

Pour cela, il suffit de terminer l'instruction PRINT par une virgule, selon le schéma :

PRINT liste d'expressions,

Dans ce cas, après avoir imprimé les valeurs des expressions de la liste, l'ordinateur rencontre le dernier symbole « virgule ». Il positionne donc, comme précédemment, la tête d'écriture au début de la zone suivante, mais comme l'instruction est terminée, il laisse la tête d'écriture à cet endroit et passe à l'exécution de l'instruction suivante.

Cette possibilité implique qu'au moment de l'exécution d'une instruction PRINT, ***la tête d'écriture peut ne pas être positionnée au début d'une ligne.***

Exemple : Considérons les deux lignes suivantes :

```
10  PRINT X1, X2, X3,
20  PRINT X4, X5
```

Après exécution de la ligne 10, la tête d'écriture est positionnée en début de quatrième zone de la ligne d'impression, en supposant qu'avant cette exécution, elle ait été positionnée au début d'une ligne.

Par conséquent, la valeur de X4 sera imprimée dans la quatrième zone de cette ligne, lors de l'exécution de la seconde instruction PRINT, et celle de X5 sera imprimée dans la cinquième zone.

Les deux lignes 10 et 20 sont en fait équivalentes, quant au résultat, à la seule ligne :

15 PRINT X1, X2, X3, X4, X5

IV.2. Impression de textes : notion de chaine

Les résultats d'un calcul imprimés sous la forme d'une succession de nombres sont le plus souvent difficiles à interpréter car il faut pouvoir associer, avec certitude, chacun de ces nombres à l'entité dont on désirait connaître la valeur. Par exemple, il faut pouvoir dire que le quatrième nombre imprimé représente le montant d'une facture alors que le septième représente un tonnage.

Lorsque le nombre de résultats est grand, ce travail sera pénible et de toute manière, ne pourra se faire qu'en connaissant parfaitement le programme et l'ordre dans lequel il imprime les résultats.

Pour éviter ces inconvénients, on a la possibilité d'imprimer sur le terminal des libellés en clair.

Un libellé en clair correspond dans l'écriture d'un programme à ce qu'on appelle une *chaîne*, c'est-à-dire une succession de caractères entourée des symboles " " (guillemets). Chacun de ces caractères peut être l'un quelconque des caractères qu'il est possible de frapper sur le terminal, y compris le blanc ⊔.

Pour imprimer un libellé, il suffit alors de faire suivre le symbole PRINT d'une chaîne.

Exemple :

10 PRINT " 1ERE ⊔ RACINE ⊔ = "

L'ordinateur imprime la succession des caractères figurant entre les guillemets, y *compris* les blancs, qui correspondent à des espacements sur le terminal.

Lors de l'impression, l'ordinateur considère une chaîne comme une « valeur » au même titre que le résultat d'une expression : il est donc possible de mélanger des chaînes et des

expressions dans une instruction PRINT. Une forme plus générale de cette instruction est donc :

PRINT liste à imprimer

Une liste à imprimer est une succession de chaînes et d'expressions séparées par des virgules.

Exemple :

```
10  PRINT " 1ERE ⊔ RACINE ⊔ = ", (3 + Y) / 8
```

Le mécanisme de l'instruction PRINT reste le même que celui que nous avons indiqué en IV.1 : à chaque rencontre d'une virgule, l'ordinateur positionne la tête d'écriture du terminal au début de la zone suivante de la ligne en cours d'impression.

Lorsque la valeur suivante à imprimer est une chaîne, il imprime la succession, dans l'ordre, des caractères de cette chaîne. Remarquons que cela peut l'amener à déborder la zone d'écriture si la chaîne comporte plus de 15 caractères.

Exemple :

```
10  PRINT " PREMIERE ⊔ RACINE ⊔ = ", (3 + Y) / 8
```

Le libellé qui comporte plus de 15 caractères, sera imprimé sur deux zones consécutives et la valeur de (3 + Y) / 8 sera imprimée dans la zone suivante.

Il existe cependant une restriction à l'impression des libellés : si la tête d'écriture du terminal arrive au bout d'une ligne, lors de l'impression d'un libellé, les caractères suivants de la chaîne correspondante seront imprimés au même endroit, c'est-à-dire qu'ils viendront tous se superposer au dernier caractère imprimé.

Dans *le cas d'impression d'un libellé, il n'y a donc pas de retour automatique à la ligne suivante* comme pour les valeurs numériques.

Exemple :

```
10  PRINT X1, X2, X3, X4,
20  PRINT " PREMIERE ⊔ RACINE ⊔ = ", (3 + Y) / 8
```

En supposant que la tête d'écriture du terminal soit en début de ligne lors de l'exécution de la première instruction PRINT, l'impression correspondant à la seconde instruction commencera dans la cinquième zone de cette ligne, car la première instruction PRINT se termine par une virgule. En conséquence, le caractère = du libellé de la deuxième instruction PRINT viendra se superposer dans l'impression au dernier E.

IV.3. Impression condensée

La disposition des valeurs imprimées dans les cinq zones d'une ligne n'est pas satisfaisante pour au moins deux raisons :

— Une valeur numérique peut être séparée du libellé qui l'identifie par un nombre important d'espaces : cas d'un libellé comportant un nombre de caractères légèrement supérieur à 15.

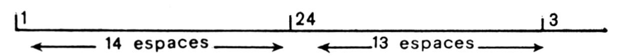
Exemple :

PREMIERE RACINE = 15.1

← 15 caractères → ← 13 espaces →

— Des valeurs numériques successives peuvent être très espacées lorsque le nombre de chiffres qu'elles comportent est petit.

Exemple :



On a donc prévu la possibilité d'une impression condensée de plusieurs valeurs sur une ligne.

Pour pallier le premier inconvénient, il est possible d'omettre dans la liste à imprimer la virgule entre une chaîne et une expression.

Dans ce cas, le libellé correspondant à la chaîne et la valeur de l'expression seront imprimés *consécutivement* sur la même ligne.

Exemple :

10 PRINT " PREMIERE ⊔ RACINE ⊔ = " (3 + Y) / 8

le résultat de cette instruction est :

|P|R|E|M|I|E|R|E| |R|A|C|I|N|E| |=| 1|5|.|1|

Pour pallier le second inconvénient, il est possible de remplacer, dans la liste à imprimer, les virgules par des points-virgules.

Lorsque deux éléments à imprimer sont séparés par un point-virgule, après l'impression du premier de ces éléments, la tête d'écriture du terminal n'est pas automatiquement positionnée au début de la zone suivante dans la ligne. Le nombre d'espacements laissé dans ce cas entre les deux valeurs imprimées dépend du système de temps partagé (ce pourra être par exemple un, deux ou trois espacements).

Remarquons que la virgule qui termine éventuellement la liste à imprimer (voir p. 45) peut, elle aussi, être remplacée par un point-virgule : l'effet est le même, à ceci près que l'ordinateur ne positionne plus la tête d'écriture au début de la zone suivante dans la ligne, mais la fait déplacer d'un certain nombre d'espaces (propres au système).

Nous reviendrons au chapitre VIII sur une application de cette possibilité.

Remarquons que dans une même liste à imprimer, peuvent figurer des virgules et des points-virgules.

Exemple :

10 PRINT "1ERE ⊔ RACINE ⊔ =", (3 + Y) / 8; (3 — Y) / 8

IV.4. Tabulation

La possibilité d'imprimer des libellés provoque des problèmes de mise en page des résultats : l'impression condensée, dans une certaine mesure, résoud ces problèmes mais manque de souplesse.

Il est souhaitable de pouvoir indiquer dans le programme l'endroit où la tête d'écriture doit être positionnée lors d'une impression.

Ceci est possible grâce à une « fonction standard » dont le sigle est TAB. Cette fonction réalise automatiquement la tabulation de la tête d'écriture au cours de l'exécution d'une instruction PRINT : elle ne peut être appelée que dans une telle instruction.

L'appel de la fonction TAB peut donc être l'un des éléments d'une liste à imprimer.

Exemple :

```
30 PRINT X ; TAB (25) ;
```

L'ordinateur imprimera la valeur de X, puis rencontrera, comme élément à imprimer, un appel de la fonction TAB, de la forme TAB (expression).

Il calculera alors la valeur de l'expression entre parenthèses ; puis déplacera la tête d'écriture du terminal jusqu'à la position correspondant à cette valeur.

Exemple : Pour l'exemple précédent, si la tête d'écriture est positionnée au début d'une ligne, avant l'exécution de l'instruction, elle se trouvera positionnée au vingt-cinquième caractère de cette ligne après l'exécution de la tabulation TAB (25).

Remarque : Lors de l'exécution d'un appel TAB (expression), il est nécessaire que la tête d'écriture soit positionnée à une valeur inférieure à celle de l'expression, sinon il n'y aura pas de tabulation.

Exemple : Supposons qu'au moment de l'exécution de l'instruction

```
10 PRINT X, TAB (10)
```

la tête d'écriture soit au début d'une ligne. La tabulation est alors inutile puisque, après l'impression de la valeur de X et la rencontre de la virgule, la tête d'écriture a été positionnée au début de la zone suivante, c'est-à-dire en position 15.

Il est donc recommandé d'entourer de points-virgules les

appels de la fonction TAB dans une instruction PRINT, pour éviter avant et après ces appels des tabulations automatiques trop grandes.

RÉSUMÉ SUR L'INSTRUCTION PRINT

La forme générale de l'instruction PRINT est la suivante :

nº de ligne PRINT liste à imprimer

La liste à imprimer est une succession de chaînes, d'expressions ou de tabulations séparées par des indicateurs de mise en page.

La liste peut être omise (voir IV.1). Elle peut aussi être terminée par un indicateur de mise en page (voir IV.2).

Les caractères des chaînes ou les valeurs des expressions de la liste sont imprimés successivement sur une même ligne ou éventuellement sur plusieurs lignes (voir IV.1). Un indicateur de mise en page réalise une tabulation automatique dans la ligne. Cet indicateur est une virgule ou un point-virgule. Il peut être omis entre une chaîne et une expression.

EXERCICES

— Quels sont les effets de la ligne

120 PRINT

— Quelles sont les trois phases de l'instruction

10 PRINT (B ↑ 2 — 4 ✕ A ✕ C)

en supposant que $A = 9$, $B = 6$ et $C = 1$

— Est-il possible d'imprimer plusieurs valeurs d'expressions sur une même ligne ? Quelle est ou quelles sont les instructions possibles pour réaliser cette impression ? Combien de valeurs peut-on placer au maximum sur la même ligne d'impression ?

— Peut-on faire imprimer directement la valeur d'une fonction standard, celle d'une fonction FNA (variable) définie auparavant ?

— Donner les instructions permettant d'imprimer un texte sur trois lignes

— Comment peut-on réaliser une mise en page lors d'une impression ?

CHAPITRE V

NOTION D'ALGORITHME
ENTRÉE DES DONNÉES EN « BASIC »

Le Programme 1 du chapitre II présente l'inconvénient d'être spécifique d'une équation du second degré particulière : celle dont les coefficients sont 4, — 3 et — 2. Si nous voulons calculer les racines d'une autre équation du second degré, nous devrons écrire un autre programme.

Dans ce cas, le travail correspondant ne serait pas important mais il n'en est pas de même lorsque le problème à traiter est plus complexe.

Dans ce chapitre, nous introduisons les notions de calculs symboliques et d'algorithme qui permettent de décrire des problèmes généraux, ou plus exactement une classe de problèmes, sans s'attacher aux valeurs particulières de chacun des problèmes de la classe.

Nous parlerons ensuite de l'analyse des problèmes, qui est un travail préparatoire fondamental à la programmation.

Nous donnerons enfin des instructions du langage BASIC qui permettent d'utiliser, effectivement, lors d'un cas pratique, un programme général qui décrit un algorithme.

V.1. CALCUL SYMBOLIQUE ET ALGORITHME

Le problème : « Calculer les racines de l'équation $4x^2 - 3x - 2 = 0$ » entre dans la classe d'un problème plus général : « Calculer les racines d'une équation du second degré. »

Dans le premier cas, le problème est indiqué de manière précise. Dans le second cas, l'expression « une équation du second degré » est un terme générique qui englobe *toutes* les équations du second degré possibles.

Il est évident que si l'on est capable de résoudre le second problème, le premier sera alors immédiatement résolu comme cas particulier où les coefficients reçoivent des valeurs numériques précises. En outre, toutes les équations avec des coefficients numériques particuliers, seront « potentiellement » résolues.

On voit l'intérêt considérable qu'il y a à donner à chacune des valeurs susceptibles de varier d'un problème à un autre, une représentation symbolique (par exemple une lettre) et de résoudre ensuite le problème avec ces représentations symboliques sans tenir compte des valeurs numériques précises qu'elles pourront avoir dans chaque cas particulier. Nous appellerons une telle façon de faire, le *calcul symbolique.*

Les représentations symboliques des valeurs susceptibles de varier d'un problème à un autre sont appelées les *paramètres* du problème.

Lorsqu'un problème a été mis sous la forme d'un calcul symbolique, il faut décrire la succession des opérations qui conduisent à la solution du problème (indépendamment de valeurs données aux paramètres). Cette description, qui doit être systématique et logique, est appelée un *algorithme.*

Remarquons, qu'on entend aussi par algorithme à la fois la description et le processus qui conduit à la solution du problème. Ainsi, un algorithme permet à une personne (ou à un ordinateur) qui ignore complètement la façon dont le problème général a été résolu, d'obtenir la solution d'un problème particulier.

Exemple : Calcul des racines d'une équation du second degré.

Le calcul symbolique correspondant concerne l'étude de l'équation :

$$ax^2 + bx + c = 0$$

où a, b et c représentent des coefficients quelconques.

Le calcul mathématique montre que l'équation a deux racines dont les expressions sont :

$$x1 = \frac{-b + \sqrt{\Delta}}{2a} \quad \text{et} \quad x2 = \frac{-b - \sqrt{\Delta}}{2a},$$

en notant Δ le discriminant $b^2 - 4ac$.

L'algorithme correspondant est donc très simple (1) :

1) Calculer le discriminant $\Delta = b^2 - 4ac$;
2) Calculer les expressions :

$$x1 = \frac{-b + \sqrt{\Delta}}{2a} \quad \text{et} \quad x2 = \frac{-b - \sqrt{\Delta}}{2a};$$

3) Imprimer les valeurs de $x1$ et $x2$.

En notant le discriminant par D au lieu de Δ (caractère qui n'existe pas en BASIC), le programme suivant décrit les étapes de l'algorithme ci-dessus :

Programme 2 :

```
10  LET D  = B ↑ 2 — 4 ⋇ A ⋇ C
20  LET X1 = (— B + SQR(D)) / (2 ⋇ A)
30  LET X2 = (— B — SQR(D)) / (2 ⋇ A)
40  PRINT X1, X2
50  END
```

Avec ce programme, l'ordinateur ne peut pas calculer et imprimer les valeurs numériques des racines puisqu'on ne lui a pas fourni de valeurs numériques pour les coefficients A, B et C.

Cependant, ce Programme 2 a l'avantage considérable sur le Programme 1 du chapitre II, de ne pas dépendre des valeurs numériques des coefficients. Ainsi, ce même programme, auquel on aura ajouté la possibilité de donner des valeurs aux variables A, B et C (voir V.3 et le chap. VIII), pourra calculer les racines de différentes équations.

(1) Nous avons volontairement laissé de côté le cas où le discriminant était négatif.

V.2. Analyse d'un problème. Organigramme

Dans la plupart des cas, un problème n'est pas donné sous la forme d'un algorithme, ni même d'un calcul symbolique : il résulte parfois de l'enchevêtrement de plusieurs méthodes données par des calculs symboliques, et plus souvent encore d'informations disparates, comprenant des processus décrits dans une langue naturelle, des formules, des méthodes de calcul, etc.

Toutes ces informations sont liées les unes aux autres selon un ordre qui n'apparaît pas toujours très clairement.

Or, comme nous l'avons vu au chapitre II, la programmation d'un problème nécessite la connaissance précise des différentes actions élémentaires (instructions) à faire exécuter à l'ordinateur, l'ordre dans lequel ces actions devront être exécutées, ainsi que celle des variables (paramètres ou résultats de calculs intermédiaires) qui apparaîtront dans le programme. Cela nécessite un travail de mise en forme du problème, pour en faciliter la programmation. Ce travail est appelé *analyse* du problème. Il consiste en :

— l'établissement d'une liste exhaustive des paramètres et l'attribution d'une variable à chacun d'eux ;
— l'établissement d'une liste exhaustive des résultats intermédiaires (intervenant à plusieurs endroits dans les calculs) et l'attribution d'une variable à chacun d'eux ;
— la décomposition du problème en une succession d'actions élémentaires et l'étude des enchaînements de ces actions.

Il est évident que plus le problème sera complexe, plus le nombre d'actions élémentaires qu'il nécessitera, sera grand. Aussi, le troisième point noté ci-dessus risque d'être particulièrement difficile. Pour tourner cette difficulté on préfère, le plus souvent, distinguer deux niveaux d'analyse : l'analyse globale et l'analyse détaillée.

La première consiste à décomposer le problème en sous-problèmes, sans entrer dans le détail des actions élémentaires de ces sous-problèmes. L'analyse a alors pour but de décrire

l'enchaînement de ces sous-problèmes ainsi que la liste de leurs paramètres et de leurs résultats.

Nous entendons par paramètre d'un sous-problème, une variable résultant d'un sous-problème qui le précède immédiatement.

Exemple : Les sous-problèmes d'un calcul d'impôt peuvent être par exemple :

— calcul du quotient familial avec pour paramètres ceux du problème général ;
— calcul de l'impôt semi-net avec pour paramètres le quotient familial et le nombre de parts ;
— calcul de la décote avec pour paramètre l'impôt semi-net et pour résultat l'impôt semi-net après décote ;
— calcul de la réduction dégressive avec pour paramètre l'impôt semi-net après décote ;
— calcul de la majoration progressive avec pour paramètre l'impôt semi-net après décote.

L'analyse détaillée, c'est-à-dire la décomposition en actions élémentaires, se fait alors au niveau des sous-problèmes.

Le résultat de l'analyse d'un problème peut être représenté graphiquement au moyen d'un schéma appelé *organigramme.*

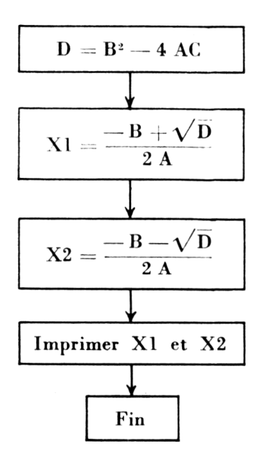
Dans ce schéma, chaque action élémentaire est décrite à l'intérieur d'un rectangle (1).

La succession dans le temps des différentes actions élémentaires est représentée en reliant les différents rectangles par des flèches.

Exemple : Organigramme de la résolution de l'équation du second degré (page ci-contre).

Remarque : La nécessité de décrire lors de l'analyse, l'enchaînement précis des actions élémentaires, n'est peut-être pas apparue suffisamment dans ce que nous avons dit précédemment. L'exemple ci-contre l'illustre clairement car il ne saurait être question de calculer X1 ou X2 avant d'avoir calculé D.

(1) Nous verrons au chapitre VII qu'il existe un autre type d'actions élémentaires, représenté d'une autre façon dans l'organigramme.



V.3. Entrée des données en BASIC

Un programme décrivant un algorithme n'est utile que dans la mesure où il permet de fournir aux paramètres des valeurs numériques : celles-ci sont appelées les *données* du programme.

En langage BASIC, il existe trois façons de fournir des données à des paramètres : nous allons voir maintenant la plus simple, les deux autres seront étudiées aux chapitres VIII et XI.

La première façon est réalisée par deux instructions :

— une description des données ;
— une lecture des données.

La première a pour seul but de décrire la liste des valeurs numériques qui seront lues comme données dans le programme. Sa forme est la suivante :

DATA liste de nombres.

Les différents nombres de la liste doivent être séparés par des virgules ou des espaces.

Exemple :

```
10 DATA 4.523, —15 ⊔ 3.5 E — 6
```

La ligne correspondant à une instruction DATA peut être placée à n'importe quel endroit d'un programme et il est possible de mettre dans un programme un nombre quelconque d'instructions DATA.

Ces instructions n'entraînent pas de calcul effectif : préalablement à l'exécution du programme, l'ordinateur extrait toutes les lignes où figure le mot DATA, et les classe par numéros de ligne croissants. Il place alors les différents nombres apparaissant dans ces lignes dans une *zone de données*, de la manière suivante :

Le premier nombre de la première ligne DATA (celle de numéro le plus petit) est placé dans la première « case » de la zone. Le nombre qui le suit, s'il existe, est placé dans la seconde « case » et ainsi de suite jusqu'à épuisement de la première ligne DATA. Ensuite, l'ordinateur place de la même façon, dans les cases suivantes les nombres de la seconde ligne DATA, puis ceux de la troisième, etc.

Exemple : Pour un programme contenant les lignes DATA suivantes :

```
 20  DATA 3, —4.5
 87  DATA 2 E 5,8
103  DATA 1
```

La zone de données est constituée par :

3	-4.5	2 E 5	8	1
case 1	case 2	case 3	case 4	case 5

Les instructions DATA ne permettent donc pas à proprement parler de fournir des valeurs numériques aux paramètres. Cette opération est réalisée par l'instruction de lecture de données dont la forme générale est :

READ liste de variables

les variables de la liste doivent être séparées par des virgules.

Exemple :

```
10  READ X, Y, Z, A, B
```

Cette instruction permet d'aller chercher une valeur numérique dans la zone de données et de l'affecter à une variable.

La valeur numérique qui sera prise est déterminée par un pointeur qui repère une case de la zone de données. Avant l'exécution du programme, ce pointeur repère la première case.

Lorsque l'ordinateur exécute une instruction READ, il affecte la valeur située dans la case *repérée* à ce moment par le pointeur, à la première variable de la liste. *Puis il déplace le pointeur d'une case vers la droite dans la zone de données.*

L'ordinateur effectue la même opération pour la variable suivante (le pointeur ne repère plus la même case puisqu'il a été déplacé d'une case vers la droite) et ainsi de suite jusqu'à épuisement des variables de la liste.

Exemple : En supposant que la zone de données est constituée par :

3	−4.5	2 E 5	8	1

et que l'instruction suivante soit la première instruction READ que l'ordinateur exécute dans le programme :

```
10  READ X, Y, Z, A, B
```

les valeurs de X, Y, Z, A et B sont respectivement 3, — 4.5, 2 E 5, 8 et 1 après exécution de cette ligne.

L'effet est donc le même que si on avait utilisé les cinq instructions d'affectation successives :

```
LET  X = 3
LET  Y = — 4.5
LET  Z = 2 E 5
LET  A = 8
LET  B = 1
```

Remarque : Si, dans une instruction READ, le pointeur a dépassé la fin de la zone de données, l'ordinateur imprime un message d'erreur et s'arrête. Ainsi, dans l'exemple précédent, si l'ordinateur trouve une autre instruction READ après la ligne 10, il imprimera un message d'erreur.

Il est possible cependant de demander à l'ordinateur de reprendre la lecture dans la zone de données à son début.

Cela est réalisé par l'instruction RESTORE, dont l'effet est de repositionner le pointeur sur la première case de la zone de données. Cette instruction présente l'intérêt d'éviter la réécriture de valeurs numériques identiques pour plusieurs paramètres.

Exemple : Supposons que les paramètres A, B, C et A1, B1, C1 doivent recevoir des valeurs numériques identiques deux à deux et que la lecture de ces deux groupes de données se fasse à deux endroits différents du programme, par exemple aux lignes 10 et 100. Au lieu de fournir six valeurs numériques dans une instruction DATA, il suffira d'en donner trois à condition de faire précéder la ligne 100 d'une instruction RESTORE :

```
 10  READ A, B, C
. . . . . . . . . .
. . . . . . . . . .
 90  RESTORE
100  READ A1, B1, C1
. . . . . . . . . . .
200  DATA 3, 7, 8.95
```

Nous pouvons maintenant compléter le Programme 2, en lui ajoutant une instruction de lecture de données qui affectera des valeurs numériques aux coefficients A, B et C.

Programme 3 :

```
10  READ A, B, C
20  LET D  = B ↑ 2 — 4 × A × C
30  LET X1 = (— B + SQR(D)) / (2 × A)
40  LET X2 = (— B — SQR(D)) / (2 × A)
50  PRINT X1, X2
60  END
```

Pour utiliser ce programme sur une équation du second degré particulière, il suffira d'y insérer une ligne DATA contenant trois valeurs numériques.

Exemple :

35 DATA 4, — 3, — 2

Pour obtenir la résolution d'une autre équation du second degré, il suffira simplement de changer les valeurs numériques de cette ligne DATA.

EXERCICES

— Qu'entend-on par algorithme, organigramme, paramètre ?

— Dresser l'organigramme du problème suivant :

Le salaire horaire d'un employé est S1, celui de sa femme est S2. Les nombres d'heures de travail par mois de l'employé et de sa femme sont respectivement N1 et N2. Calculer et imprimer les revenus mensuels de l'employé et de sa femme ainsi que le revenu total du ménage.

Ecrire la séquence d'instructions traduisant l'organigramme en supposant de plus que S1 est égal à 9,50 F, S2 à 6,30, N1 à 180 h et N2 à 120 h.

— Quelle est l'instruction associée à une instruction de lecture READ ?

— Peut-il y avoir plus de variables dans la liste suivant READ que de données dans la zone créée par les instructions DATA ? et réciproquement ?

— Quel est l'effet de l'instruction RESTORE ?

CHAPITRE VI

BOUCLES DE CALCUL ET RUPTURES DE SÉQUENCES

Dans les chapitres précédents, tous les calculs présentés sont une succession d'actions élémentaires se déroulant l'une après l'autre du début à la fin sans aucun retour possible en arrière. De tels calculs sont dits *séquentiels*.

L'intérêt d'un langage de programmation serait faible s'il ne permettait d'effectuer que des calculs séquentiels.

En effet, la plupart des problèmes qui se présentent dans la pratique contiennent des calculs qui doivent être effectués plusieurs fois successives, bien souvent en ne changeant d'une fois à l'autre que les valeurs des variables intervenant dans ces calculs.

Par exemple, la paie dans une entreprise est un problème qui doit être effectué pour chacun des salariés ; le principe du calcul reste le même (algorithme), mais le nombre d'heures de travail, le salaire horaire, la qualification..., changent d'un salarié à l'autre.

Un autre cas, se présentant très fréquemment, est celui de l'itération, c'est-à-dire d'un calcul qui doit être recommencé à partir des valeurs fournies par l'exécution précédente de ce même calcul.

Cela nous conduit naturellement à la notion de boucle de calcul, c'est-à-dire d'un calcul qui est exécuté plusieurs fois de suite.

Dans ce chapitre, nous examinerons d'abord comment il est possible de réaliser de manière simple de telles boucles de calcul au moyen d'une instruction particulière. Nous verrons aussi les limites des possibilités de cette instruction et nous en présenterons une autre qui la généralise dans une certaine mesure.

VI.1. Ruptures de séquence inconditionnelles

Reprenons l'exemple de l'équation du second degré. Supposons que nous voulions faire calculer successivement, par un *même* programme, les racines de *plusieurs* équations sans modifier à chaque fois la ligne DATA de ce programme.

On pourrait penser à utiliser le Programme 3 du chapitre V, en insérant une ligne DATA contenant plus de trois valeurs (en fait $3n$ valeurs si on veut résoudre n équations).

Exemple :

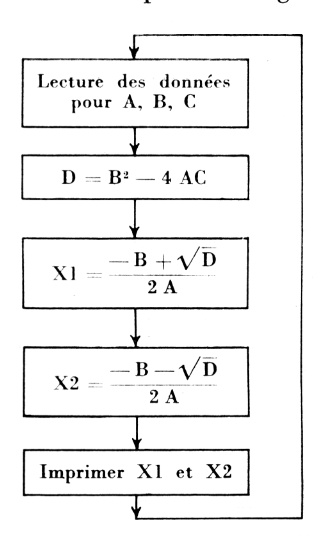
```
10  READ A, B, C
20  LET D  = B ↑ 2 — 4 ⋇ A ⋇ C
30  LET X1 = (— B + SQR (D)) / (2 ⋇ A)
40  LET X2 = (— B — SQR (D)) / (2 ⋇ A)
50  PRINT X1, X2
55  DATA 4, — 3, — 2, 7, — 8, — 1.5, 4, 2, — 6
60  END
```

Cependant, nous n'obtiendrons pas le résultat escompté car après avoir lu les trois premières valeurs de la zone de données, et les avoir affectées à A, B, C, l'ordinateur calculera et imprimera bien les racines de la première équation, mais après la ligne 50, l'instruction qu'il exécutera sera celle de la ligne 60 (la ligne 55 n'entraîne pas de calcul). L'ordinateur s'arrêtera donc sans avoir calculé les racines des équations suivantes.

Pour éviter cet arrêt dès la résolution de la première équation, nous indiquons à l'ordinateur à la fin de cette première résolution, qu'il doit recommencer les calculs qu'il vient de faire en affectant aux variables A, B et C de nouvelles valeurs.

L'action élémentaire qui suit l'impression des racines ne

doit donc plus être la fin du programme, mais la lecture de nouvelles données. Cela correspond à l'organigramme suivant :



La flèche qui « remonte » de la dernière opération élémentaire « imprimer X1 et X2 » vers la première « lecture des données » correspond à une action élémentaire que l'ordinateur effectuera, et qu'il faut lui indiquer explicitement.

Cela se fait au moyen d'une instruction GØTØ dont la forme générale est :

GØTØ numéro de ligne.

Lorsque l'ordinateur rencontre au cours de l'exécution du programme, une telle instruction à une ligne de numéro n (par exemple $n = 100$), il n'exécute pas alors la ligne de numéro immédiatement supérieur à n, mais celle dont le numéro est indiqué après GØTØ.

Dans l'exemple précédent, il suffit de mettre après la ligne 50 (impression des racines X1 et X2) une instruction :

GØTØ 10.

Cette instruction doit être placée avant la ligne 60, sinon après la ligne 50, l'ordinateur exécuterait la ligne 60, c'est-à-dire qu'il s'arrêterait.

Programme 4 :

```
10  READ A, B, C
20  LET D  = B ↑ 2 — 4 × A × C
30  LET X1 = (— B + SQR (D)) / (2 × A)
40  LET X2 = (— B — SQR (D)) / (2 × A)
50  PRINT X1, X2
52  GØTØ 10
55  DATA 4, — 3, — 2, 7, — 8, — 1.5, 4, 2, — 6
60  END
```

Lorsque l'ordinateur a imprimé les racines de la première équation (ligne 50), il exécute la ligne 52, donc il « revient en arrière » pour exécuter de nouveau la ligne 10, puis la ligne 20, etc.

Comme à ce moment, le pointeur de la zone de données repère la valeur 7, la seconde exécution de la ligne 10 affectera à A, B et C respectivement les valeurs 7, — 8 et — 1.5 (la première fois, les valeurs affectées à A, B et C ont été 4, — 3 et — 2). Le second calcul des lignes 20, 30 et 40 s'effectuera donc avec ces nouvelles valeurs des coefficients A, B et C.

Remarques :

— une instruction GØTØ avec comme numéro de ligne, celui de la dernière ligne du programme (instruction END) peut être remplacée par le seul mot STØP. Ceci est particulièrement utile lors de l'écriture d'un programme dont on ne connaît pas déjà le numéro de la dernière ligne.

— une instruction de la forme

```
10  GØTØ 11
```

est inutile puisque, de toute manière, l'ordinateur exécute la ligne 11 après la ligne 10 (si bien sûr il existe une ligne 11).

Toute ligne dont le numéro suit immédiatement celui d'une ligne où figure une instruction GØTØ, n'est « accessible » (1) qu'à partir d'une ligne où figure une autre instruction GØTØ. En d'autres termes, une telle instruction ne sera exécutée que si une autre instruction s'y renvoie.

Exemple :

```
10  READ A, B, C
20  LET D  = B ↑ 2 — 4 × A × C
30  LET X1 = (— B + SQR (D)) / (2 × A)
40  LET X2 = (— B — SQR (D)) / (2 × A)
50  PRINT X1, X2
52  GØTØ 10
53  PRINT D
55  DATA 4, — 3, — 2
60  END
```

Dans cet exemple, l'instruction de la ligne 53 ne sera jamais exécutée car aucune autre instruction ne s'y renvoie.

Pour l'exécuter, il faut insérer une instruction GØTØ, avant la ligne 52, par exemple :

```
25  GØTØ 53
```

Remarquons qu'avec le programme modifié de cette façon, l'ordinateur exécutera dans l'ordre les instructions des lignes 10, 20, 25, 53 et 60, c'est-à-dire qu'il imprimera uniquement le discriminant de la première équation (ligne 53).

Pour calculer les racines et les imprimer, il est nécessaire d'ajouter une instruction GØTØ après la ligne 53 et avant la ligne 60.

```
10  READ A, B, C
20  LET D  = B ↑ 2 — 4 × A × C
25  GØTØ 53
30  LET X1 = (— B + SQR (D)) / (2 × A)
40  LET X2 = (— B + SQR (D)) / (2 × A)
50  PRINT X1, X2
```

(1) Nous verrons au § VI.2 et au chap. VII d'autres possibilités d'accéder à une telle ligne.

```
52 GØTØ 10
53 PRINT D
54 GØTØ 30
55 DATA 4, — 3, — 2
60 END
```

Ce programme n'est bien sûr pas à prendre pour modèle. On obtiendrait le même résultat en supprimant les lignes 25 et 54 et en remplaçant la ligne 53 par une ligne :

```
25 PRINT D.
```

Nous l'avons donné pour décrire les difficultés d'utilisation de l'instruction GØTØ.

Insistons encore sur le fait qu'elle correspond dans l'organigramme à une flèche qui joint deux actions élémentaires ne se trouvant pas l'une au-dessous de l'autre. Cette instruction est indispensable mais insuffisante pour décrire la notion de boucle. En effet, elle ne permet pas de contrôler le nombre de fois où la boucle est exécutée.

Reprenons l'exemple du Programme 4 : après avoir exécuté pour la troisième fois l'instruction de la ligne 50, donc après avoir imprimé les racines de la troisième équation, l'ordinateur exécute l'instruction de la ligne 52, c'est-à-dire qu'il revient à la ligne 10 pour lire de nouvelles données. Mais à ce moment, le pointeur de la zone de données a dépassé la dernière valeur (— 6) de cette zone et l'ordinateur ne peut plus rien y lire. Il imprime un message d'erreur et s'arrête.

Remarquons que si une telle erreur ne s'était pas manifestée, l'ordinateur aurait continué « indéfiniment ».

Il est donc indispensable de pouvoir contrôler le nombre de fois où une boucle de calculs est effectuée : nous donnons maintenant une première façon d'effectuer ce contrôle en langage BASIC.

VI.2. Ruptures de séquence calculées

Supposons que nous connaissions à l'avance le nombre d'équations à résoudre et que ce nombre soit limité ; par exemple, trois équations. La boucle de calcul des racines doit

donc être effectuée trois fois et pour le contrôler, nous utiliserons une variable I dont nous ferons progresser la valeur de 1 chaque fois que la boucle a été effectuée. Une telle variable est appelée un *compteur*.

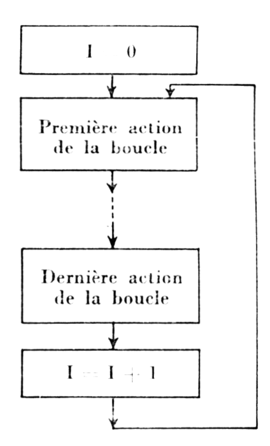
Pour que la valeur de ce compteur représente bien le nombre de fois où la boucle a été effectuée, il faut :

— lui donner une valeur initiale nulle en dehors de la boucle, c'est-à-dire avant la première action élémentaire de cette boucle ;
— le faire progresser de 1 à la fin de la boucle, c'est-à-dire après la dernière opération élémentaire de la boucle.

La première opération est réalisée par l'action élémentaire I = 0. La seconde nécessite d'ajouter 1 à la précédente valeur du compteur, c'est-à-dire à la variable I, et de prendre le résultat de cette addition pour nouvelle valeur du compteur ; c'est-à-dire d'affecter ce résultat à la variable I. L'action élémentaire correspondante est donc :

$$I = I + 1.$$

L'organigramme ci-dessous schématise ce que nous venons de dire :

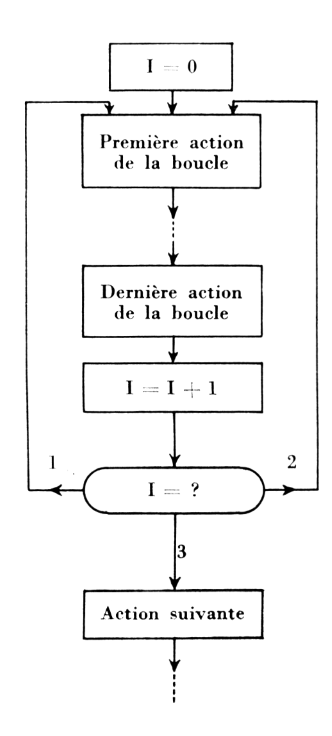


Cet organigramme ne teste cependant pas la valeur du compteur et ne permet donc pas d'arrêter l'exécution de la boucle.

Il convient de le compléter par une nouvelle action élémentaire qui suivra l'action $I = I + 1$ et qui indiquera s'il y a lieu ou non de continuer la boucle.

Par exemple, si la valeur de I est égale à 1 ou 2, la boucle est effectuée une fois de plus et si elle est égale à 3, la boucle est « terminée ». Dans ce dernier cas, l'action élémentaire qui suit est la première à effectuer après la boucle.

Ce test, action élémentaire particulière qui différencie les traitements ultérieurs suivant la valeur de I, est représenté dans l'organigramme de la manière suivante :



L'expression dont il faut tester la valeur, suivie d'un signe = et d'un point d'interrogation, est placée dans un ovale. De cet ovale, partent *plusieurs* flèches (au moins deux, contrairement aux autres actions élémentaires). Sur chacune d'elles, on indique une valeur possible pour la quantité à tester, et la flèche correspondante conduit à l'action élémentaire suivante que cette valeur détermine.

En BASIC, une instruction correspond à cette action élémentaire spéciale : la rupture de séquence calculée ou, plus brièvement, GØTØ calculé. Elle permet d'indiquer à l'ordinateur la ligne contenant l'instruction à exécuter selon la valeur d'une expression.

Exemple :

```
53  ØN I GØTØ 10, 10, 60
```

Sa forme la plus générale est la suivante :

ØN expression GØTØ liste de numéros de ligne.

Dans la liste, les numéros de ligne sont séparés par des virgules.

Pour exécuter cette instruction, l'ordinateur calcule d'abord la valeur de l'expression après le symbole ØN (si cette valeur n'est pas un nombre entier, il l'arrondit au nombre entier le plus proche). Notons n cette valeur. L'ordinateur exécute alors l'instruction dont le numéro de ligne est le n-ième dans la liste après GØTØ.

Nous pouvons alors modifier le Programme 4, de manière à éviter qu'il s'arrête sur une erreur.

Programme 5 :

```
 5  LET I   = 0
10  READ A, B, C
20  LET D   = B ↑ 2 — 4 × A × C
30  LET X1 = (— B + SQR (D)) / (2 × A)
40  LET X2 = (— B — SQR (D)) / (2 × A)
50  PRINT X1, X2
52  LET I = I + 1
53  ØN I  GØTØ 10, 10, 60
```

```
55  DATA 4, — 3, — 2, 7, — 8, — 1.5, 4, 2, — 6
60  END
```

Remarque : Dans la plupart des systèmes de temps partagé, les valeurs des variables en BASIC sont initialisées à 0. La ligne 5 est donc inutile. Cependant, cette facilité est dangereuse, car il arrive que les variables ne soient pas automatiquement réinitialisées à 0 après une première exécution d'un programme. Une seconde exécution de ce même programme risque de produire des résultats faux.

Dans le cas où nous l'avons présentée, l'instruction GØTØ calculée manque de souplesse. Elle suppose en effet connu à l'avance le nombre d'équations à résoudre et surtout elle limite ce nombre, puisque tous les numéros des lignes de la liste doivent être écrits sur une seule ligne.

Nous verrons au chapitre VII, une façon plus souple de contrôler le nombre de fois où une boucle est effectuée.

Cependant, l'instruction GØTØ calculée est très utile et particulièrement simple d'emploi, dans le cas où un calcul doit être nettement différencié suivant la valeur d'une expression.

Exemple : Le calcul du salaire d'un employé dépend de son état civil E, codé comme suit : célibataire = 1, marié = 2, veuf = 3.

Pour chaque employé, la valeur de E est lue dans une zone de données.

Suivant cette valeur, on imprime les libellés « célibataire », « marié » ou « veuf » et on effectue les traitements 1, 2 et 3 (que nous ne détaillerons pas).

Le programme pourra alors être le suivant :

```
 10  READ E
 20  ØN E  GØTØ 30, 100, 200
 30  PRINT " CELIBATAIRE "
     « Traitement 1  »
100  PRINT " MARIE "
     « Traitement 2 »
200  PRINT " VEUF "
     « Traitement 3 »
```

EXERCICES

— Quelle est l'utilité d'un compteur dans un programme ?

— Reprendre le second exercice à la fin du chapitre V. Introduire les mêmes éléments pour deux autres ménages. Compléter l'organigramme tracé et le programme qu'il était demandé d'écrire. Utiliser un compteur du nombre de ménages ; on s'arrêtera lorsque le nombre de ménages sera égal à 3.

— Avec les instructions données dans ce chapitre, peut-on calculer et imprimer la racine carrée des N premiers nombres entiers quelle que soit la valeur de N ?

CHAPITRE VII

CALCULS ITÉRATIFS

Nous avons indiqué au chapitre précédent une façon de contrôler le nombre de fois où une séquence de calculs est effectuée. Nous avons dit également que cette manière de procéder manquait de souplesse.

En fait, pour effectuer ce contrôle, il suffit de tester si la valeur du compteur a, ou non, dépassé une valeur donnée. Deux réponses sont possibles : oui ou non, c'est-à-dire que le test est vérifié ou ne l'est pas.

Cette possibilité se rencontre constamment dans les problèmes pratiques : ainsi, dans les Programmes 2 à 5 des chapitres V et VI, nous avons (volontairement) ignoré le cas où le discriminant D était négatif. Cela peut conduire à une erreur puisque aux lignes 20 et 30 de ces programmes, on prend la racine carrée de D. Il convient donc de tester si la valeur de D est positive ou non avant d'en extraire la racine carrée.

Nous allons voir dans ce chapitre qu'il est possible d'effectuer de tels tests en BASIC. Nous verrons ensuite une simplification de l'écriture de certaines boucles, particulièrement agréable à l'emploi et à la lecture dans un programme.

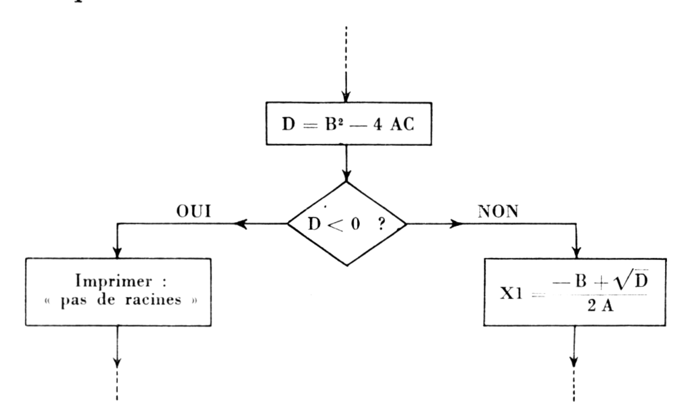
VII.1. RUPTURE DE SÉQUENCE CONDITIONNELLE

Un test permet de savoir si une condition est réalisée ou non. Si la condition est réalisée, le traitement est orienté vers une certaine succession d'actions élémentaires ; si la condition n'est pas réalisée, le traitement est orienté vers une autre succession d'opérations élémentaires.

Un test est donc une opération élémentaire au même titre que celles que nous avons déjà vues. Cependant, il lui correspond dans l'organigramme une représentation spéciale puisque le test est suivi de deux actions élémentaires. (L'exécution de l'une ou de l'autre dépend de la réponse du test.)

La condition suivie d'un point d'interrogation est placée dans un losange. De ce losange partent deux flèches. Sur l'une d'elles, on marque OUI, sur l'autre NON. Chacune des flèches conduit à l'action élémentaire suivante selon la réponse du test.

Exemple :



En langage BASIC, les conditions qui peuvent être testées sont de la forme suivante :

expression 1 opérateur de relation expression 2.

Les opérateurs de relations possibles sont :

$=$, $<>$ (différent), $>$, $>=$ (supérieur ou égal) ;
$<$, $<=$ (inférieur ou égal).

Pour savoir si la condition est réalisée, l'ordinateur calcule les valeurs de l'expression 1 et de l'expression 2, puis compare ces deux valeurs en fonction de l'opérateur de relation indiqué.

Exemples :

— B ↑ 2 — 4 * A * C > 0. Cette condition est réalisée si la valeur de l'expression B ↑ 2 — 4 * A * C est positive ;
— A < > 0. Cette condition est réalisée si la valeur de A est différente de 0 ;
— I = 3. Cette condition est réalisée si la valeur de I est égale à 3 ;
— SQR (1 + X ↑ 2) * Y > = SIN (SQR (1 — Y ↑ 2)) ↑ 2. Cette condition est réalisée si la valeur de la première expression est supérieure ou égale à celle de la seconde expression.

Le test d'une condition s'écrit très simplement, au moyen de la rupture de séquence conditionnelle (ou plus simplement instruction conditionnelle), qui est une instruction de la forme :

IF condition THEN numéro de ligne.

En langage clair, elle signifie :

— si la condition est réalisée, alors exécuter l'instruction située à la ligne dont le numéro est indiqué, sinon, exécuter la ligne qui suit immédiatement l'instruction conditionnelle.

L'effet d'une telle instruction est donc le suivant :

— l'ordinateur examine la condition ;
— si elle est réalisée, la ligne qu'il exécute ensuite est celle dont le numéro est indiqué après THEN ;
— si elle n'est pas réalisée, la ligne qu'il exécute est, comme pour les autres instructions (sauf GØTØ), celle dont le numéro est immédiatement supérieur au numéro de la ligne qu'il vient d'exécuter.

Exemple :

```
50  IF D > = 0 THEN 80
60  PRINT " PAS DE RACINES "
70  STØP
80  LET X1 = (— B + SQR (D)) / (2 * A)
 .
 .
 .
```

Après l'exécution de la ligne 50, l'ordinateur exécute la ligne 80 si D est positif ou nul et la ligne 60 si D est négatif.

Remarques :

— Comme pour l'instruction GØTØ (voir chap. VI), une ligne telle que :

10 IF D > 0 THEN 11

n'a pas de sens. En effet, dans tous les cas (que D soit positif ou non) la ligne 11 sera exécutée après la ligne 10.

— Le branchement explicite après THEN n'est pas indiqué au moyen du symbole GØTØ, mais uniquement à l'aide d'un numéro de ligne. Une ligne telle que :

10 IF D > 0 THEN GØTØ 50

est incorrecte.

— Lorsqu'une condition de la forme
expression 1 < expression 2
n'est pas réalisée, alors la valeur de l'expression 1 est supérieure ou *égale* à celle de l'expression 2 : si la condition D < 0 n'est pas réalisée, alors la valeur de D est positive ou *nulle*.

La même remarque vaut pour l'opérateur >.

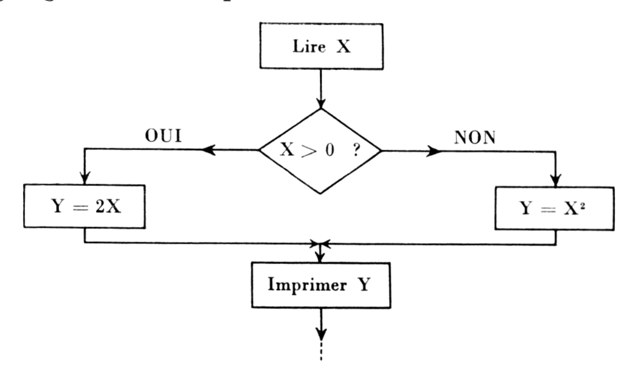
VII.2. Instruction conditionnelle et instruction GØTØ

Il peut sembler que l'existence de l'instruction conditionnelle permette de se passer de l'instruction GØTØ. En fait, il n'en est rien comme le montre l'exemple suivant :

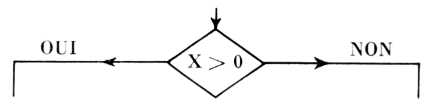
Exemple : Lire une variable X.

— si sa valeur est positive, affecter à une variable Y le double de X, puis imprimer la valeur de Y ;
— si sa valeur est négative ou nulle, affecter à la variable Y le carré de X, puis imprimer la valeur de Y.

En outre, la suite des calculs est identique dans les deux cas. L'organigramme correspondant sera donc le suivant :



Les traitements se « regroupent » après avoir été différenciés par le test :



Examinons le programme qui réalise cet organigramme.

Le test correspond à une instruction conditionnelle, par exemple de la forme :

10 IF X > 0 THEN un numéro de ligne

Nous ne précisons pas pour l'instant le numéro de ligne, car nous ne savons pas à quelle ligne nous décrirons l'action élémentaire $Y = 2X$ qui suit le test lorsque celui-ci a répondu OUI. Remarquons que ceci revient, lors de l'écriture du programme, à laisser provisoirement de côté la partie de l'organigramme correspondant au cas où le test a répondu OUI.

Au contraire, la ligne de numéro suivant, par exemple 20, décrira l'action élémentaire qui suit le test lorsque celui-ci a répondu NON. Donc, après une instruction conditionnelle, on décrit la succession d'instructions correspondant à la « branche NON » de l'organigramme.

Dans l'exemple actuel, ce sera :

```
20  LET Y = X ↑ 2
30  PRINT Y
40  END
```

Reste à décrire la branche OUI du test et, en particulier, sa première action élémentaire Y = 2X.

La ligne correspondante doit avoir un numéro *n* inférieur à 40 puisque celui-ci est le numéro de la dernière ligne du programme, et d'autre part ce numéro doit être supérieur à 10.

Examinons les différentes valeurs possibles pour *n* :

— *n* peut-il être inférieur à 20 ? évidemment non, puisque dans ce cas l'instruction de la ligne 10 ne sert à rien ;

— *n* peut-il être compris entre 20 et 30 ; par exemple 25 ? dans ce cas on obtient le programme suivant :

```
10  IF X > 0  THEN 25
20  LET Y = X ↑ 2
25  LET Y = 2 * X
30  PRINT Y
40  END
```

Lorsque X est positif, on obtient bien le résultat escompté, mais on ne l'obtient plus lorsque X est négatif ou nul. En effet, le test de la ligne 10 ayant répondu NON, l'ordinateur exécute la ligne 20, puis la ligne 25 et la ligne 30, c'est-à-dire que c'est la valeur de 2 * X qui est affectée finalement à Y et non celle de X^2 ;

— *n* peut-il être compris entre 30 et 40 ? le même raisonnement que ci-dessus aboutit encore à un résultat non désiré lorsque X est négatif ou nul.

Tous les cas sont donc à rejeter. Cela provient d'une erreur dans la programmation initiale. Lorsque nous avons décrit la branche NON de l'organigramme, nous aurions dû nous arrêter, lorsque cette branche se regroupe avec d'autres branches.

Ce regroupement correspond à une rupture de séquence implicite vers une succession d'actions élémentaires communes à plusieurs branches.

Il faut donc remplacer la ligne 30 par une instruction GØTØ

vers un numéro de ligne (encore inconnu) où sera décrite la première action élémentaire du regroupement, c'est-à-dire « imprimer Y ».

L'écriture du programme commencera donc de la manière suivante :

```
10  IF X > 0  THEN numéro de ligne encore inconnu
20  LET Y = X ↑ 2
30  GØTØ numéro de ligne encore inconnu
```

Arrivé à ce stade de l'écriture, comme nous l'avons déjà dit, on ne peut accéder à la ligne qui suit la ligne 30 que par une instruction GØTØ (ou encore par une instruction conditionnelle). Cette ligne que nous pouvons considérer comme « libre » dans l'état actuel de l'écriture nous permet de décrire la partie de l'organigramme laissée en suspens, c'est-à-dire la branche OUI.

Nous écrirons donc à partir de cette ligne la succession des actions élémentaires de cette branche. Dans l'exemple pris, il n'y en a qu'une Y = 2X que nous écrirons à la ligne suivante :

```
40  LET Y = 2 * X
```

Remarquons que nous pouvons maintenant compléter l'écriture de la ligne 10 : en effet, nous y avions laissé inconnu le numéro de ligne à exécuter quand X est positif. Ce numéro est connu maintenant : il est égal à 40.

Le programme se complète ainsi :

```
10  IF X > 0  THEN 40
20  LET Y = X ↑ 2
30  GØTØ « numéro de ligne encore inconnu »
40  LET Y = 2 * X
```

La ligne suivante doit décrire l'action élémentaire qui suit Y = 2X dans la branche OUI de l'organigramme. Cette action est un branchement vers la première action élémentaire qui suit le « regroupement ».

Ainsi, on peut continuer à décrire à partir de la ligne suivante la succession des opérations élémentaires du regroupement. En particulier, cette ligne suivante, de numéro 50 par exemple, décrira la première action élémentaire du regrou-

pement. Il est maintenant possible de compléter la ligne 30.
Nous obtenons finalement l'écriture suivante :

```
10  IF X > 0  THEN 40
20  LET Y = X ↑ 2
30  GØTØ 50
40  LET Y = 2 * X
50  PRINT Y
.
.
.
```

Cet exemple montre que l'instruction GØTØ est indispensable ; sans elle, nous n'aurions pas pu décrire la rupture de séquence inconditionnelle de la ligne 30.

Cet exemple donne aussi une façon systématique d'écrire un programme à partir d'un organigramme où figurent des tests : Le programme doit d'abord décrire la branche NON de l'organigramme (ou la succession des branches NON en cas de tests successifs : voir plus loin), jusqu'à la dernière action élémentaire avant le regroupement de plusieurs branches. Ce regroupement correspond à une instruction GØTØ à un numéro de ligne « laissé en attente ».

Puis il doit décrire après cette instruction GØTØ, la branche OUI du dernier test rencontré précédemment, de la même façon que ci-dessus, et ainsi de suite jusqu'à l' « épuisement » de toutes les branches.

Seule la partie du programme qui décrit la dernière branche « explorée » (celle où tous les tests ont répondu OUI) n'a pas besoin d'être suivie d'une instruction GØTØ. La description du regroupement général doit au contraire suivre immédiatement cette partie de programme et il est alors possible de compléter toutes les instructions GØTØ laissées en attente.

La possibilité de tester des conditions accroît considérablement la puissance des traitements que l'ordinateur peut effectuer ; elle accroît aussi la complexité des organigrammes, car, un test étant une action élémentaire comme une autre, il est possible d'effectuer successivement autant de tests que l'on veut.

Exemple : Supposons qu'il faille effectuer des calculs différents suivant la position d'une variable X par rapport aux nombres 0, 1 et 10.

Prenons le cas suivant :

Si X est négatif, affecter X à une variable Y et 2X à une variable Z.

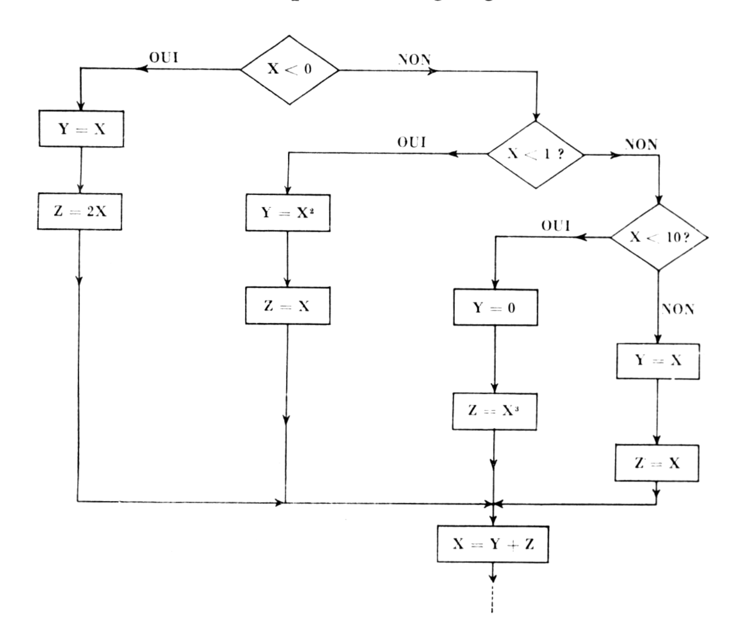
Si X est positif ou nul, mais inférieur à 1, affecter X^2 à Y et X à Z.

Si X est supérieur ou égal à 1, mais inférieur à 10, affecter 0 à Y et X^3 à Z.

Enfin, si X est au moins égal à 10, affecter X à Y et à Z.

Après avoir effectué ces différents traitements, affecter à X la somme des valeurs obtenues pour Y et Z.

Ce traitement correspond à l'organigramme suivant :



Avec ce que nous avons dit précédemment, le programme correspondant à cet organigramme sera le suivant :

```
 10  IF X <  0  THEN 130
 20  IF X <  1  THEN 100
 30  IF X < 10  THEN  70
 40  LET Y = X
 50  LET Z = X
 60  GØTØ 150
 70  LET Y = 0
 80  LET Z = X ↑ 3
 90  GØTØ 150
100  LET Y = X ↑ 2
110  LET Z = X
120  GØTØ 150
130  LET Y = X
140  LET Z = 2 * X
150  LET X = Y + Z
  .
  .
  .
```

VII.3. Cas de tests successifs

Lorsqu'un problème comporte de nombreux tests successifs, il n'est pas toujours simple d'en trouver l'organigramme.

Le premier travail d'analyse est alors de dresser la liste exhaustive de tous les éléments sur lesquels porteront les tests : ces éléments sont de deux sortes :

— des paramètres du problème ;
— des variables résultats de calculs intermédiaires ou des expressions.

Il faut ensuite étudier et décrire précisément l'enchaînement chronologique de tous les tests qui portent sur ces éléments. Donnons une méthode qui facilite le travail.

Cette méthode, dite d'enchaînement logique de tests, consiste à décomposer le traitement global en sous-traitements, en fonction des différents tests.

Chaque test décompose et oriente un sous-traitement donné en deux nouveaux sous-traitements. Chacun d'eux est à son tour décomposé en fonction des tests qu'il peut comprendre.

Un sous-traitement est simplement décrit par un terme générique. Le détail de ses actions élémentaires n'est pas explicité. Cette décomposition est recommencée jusqu'à ne plus obtenir que des sous-traitements ne comprenant aucun test et jusqu'à ce que tous les sous-traitements définis dans la méthode et comprenant des tests aient été décomposés.

Exemple : résolution complète de $AX^2 + BX + C = 0$.

— Si A est nul, cette équation est du premier degré et n'a, alors, de racine (unique) que si B est non nul ; si B est nul, l'équation n'a pas de sens si C est non nul, et elle est indéterminée si C est nul ;
— Si A est non nul, l'équation est du second degré. Elle possède deux racines si son discriminant D est positif. Elle en possède une si D est nul et aucune si D est négatif.

Liste des éléments à tester :

— paramètres : A, B, C ;
— expressions : D.

Enchaînement logique des tests (nous indiquons un sous-traitement par les initiales ST) :

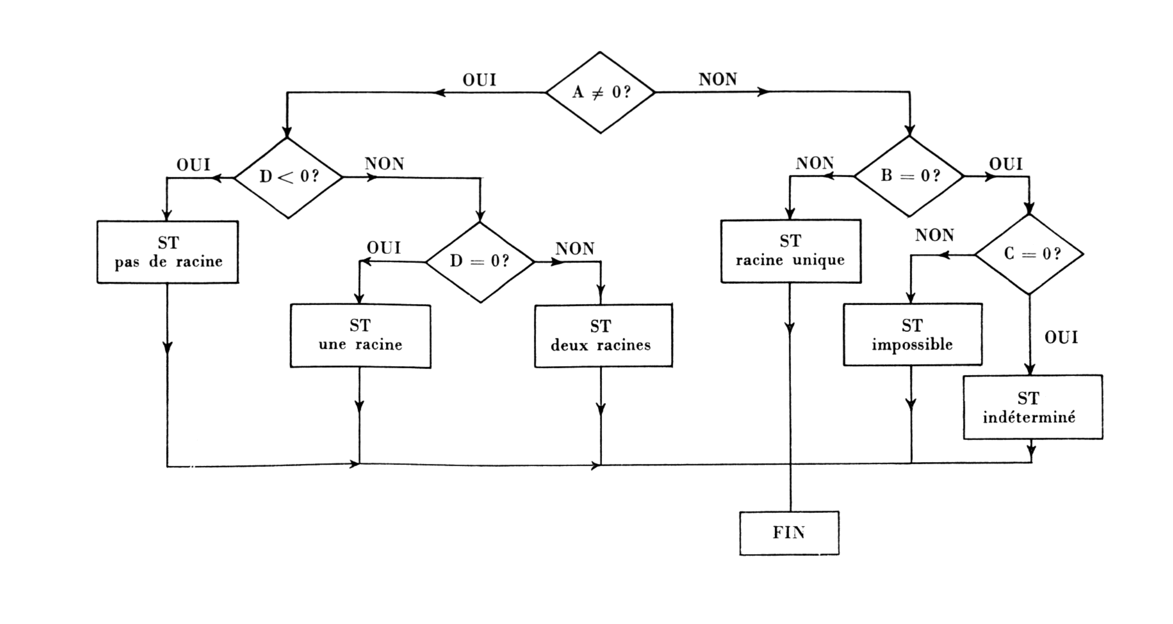
$A \neq 0$?
OUI = ST second degré
NON = ST premier degré.

ST second degré :

$D < 0$?
OUI = ST pas de racine
NON = ST racines.

ST racines :

$D = 0$?
OUI = ST une racine
NON = ST deux racines.


A ≠ 0?
OUI
NON
D < 0?
OUI
NON
ST
pas de racine
D = 0?
OUI
NON
ST
une racine
ST
deux racines
B = 0?
NON
OUI
ST
racine unique
C = 0?
NON
OUI
ST
impossible
ST
indéterminé
FIN

ST premier degré :

B = 0 ? OUI = ST erreur
NON = ST racine unique.

ST erreur :

C = 0 ? OUI = ST indéterminé
NON = ST impossible.

Les sous-traitements ST pas de racines, ST une racine, ST deux racines, ST racine unique, ST indéterminé et ST impossible ne comportent plus de tests.

Cette méthode permet de construire immédiatement l'organigramme global ci-contre.

Il suffit ensuite de détailler dans des organigrammes particuliers les différents sous-traitements ne comportant pas de test (à chacun de ces sous-traitements correspondra une partie de programme sans instruction conditionnelle). Nous ne détaillerons pas ces organigrammes qui ne présentent aucune difficulté.

Le programme général décrit à partir de ces organigrammes est le suivant :

Programme 6 :

```
 10  READ A, B, C
 20  IF A < > 0  THEN 110
 30  IF B = 0  THEN 60
 40  PRINT " 1ER DEGRE : RACINE = " — C/B
 50  STØP
 60  IF C = 0  THEN 90
 70  PRINT " IMPØSSIBLE "
 80  STØP
 90  PRINT " INDETERMINE "
100  STØP
110  IF D < 0  THEN 190
120  IF D = 0  THEN 170
130  LET X1 = (— B + SQR (D)) / (2 * A)
```

```
140  LET X2 = (— B — SQR (D)) / (2 × A)
150  PRINT " 2D DEGRE : 1ERE RACINE = " X1 ; " 2DE RACINE = " X2
160  STØP
170  PRINT " 2D DEGRE : UNE RACINE = " — B / (2 × A)
180  STØP
190  PRINT " 2D DEGRE : PAS DE RACINES "
200  END
```

Nous n'avons pas fait figurer dans ce programme de ligne DATA ; nous n'avons pas non plus prévu la résolution de plusieurs équations. Nous pouvons y remédier simplement en remplaçant toutes les lignes où figurent une instruction STØP par un GØTØ calculé et en intercalant entre les lignes 190 et 200 un GØTØ calculé.

Mais nous pouvons aussi procéder autrement comme nous allons le voir maintenant.

VII.4. Controle de boucle. Boucle POUR

Un test, et l'instruction conditionnelle qui lui correspond, permet de contrôler facilement le nombre de fois où une boucle de calcul est effectuée. Il suffit, en effet, de comparer la valeur du compteur I après la dernière action de la boucle, à une valeur N donnée à l'avance, nombre de fois où la boucle doit être effectuée (N peut évidemment être un paramètre du problème).

Si I est inférieur à N, il faut recommencer la boucle, sinon il faut exécuter la première action qui suit la boucle. Nous obtenons donc l'organigramme ci-contre.

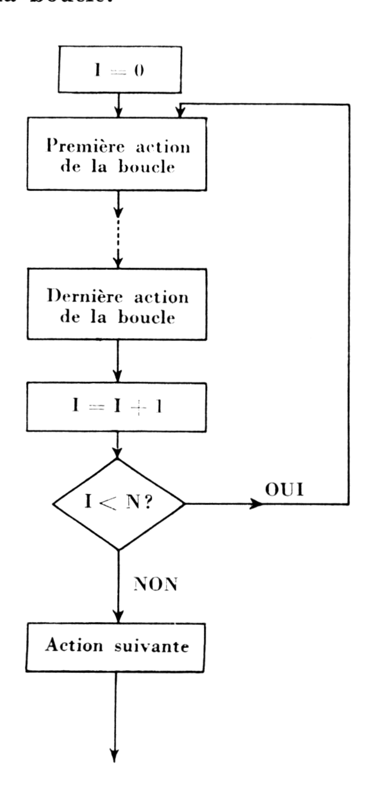
Le Programme 5 du chapitre VI peut donc encore s'écrire :

Programme 6 :

```
5  LET I = 0
7  READ N
```

```
10 )
à  } lignes identiques à celles du Programme 5
52 )
53  IF I < N  THEN 10
54  DATA 3
55  DATA 4, — 3, — 2, 7, — 8, — 1.5, 4, 2, — 6
60  END
```

Remarquons que le nombre N de fois où la boucle sera effectuée est donné à la ligne 54 et qu'il est lu avant la première instruction de la boucle.



Autre exemple : Calculer et imprimer les 10 premiers nombres entiers et leurs racines carrées.

Programme 7 :

```
10  LET I = 1
20  PRINT I, SQR (I)
30  LET I = I + 1
40  IF I < = 10  THEN 20
50  END
```

Ce programme peut être écrit de façon plus courte et plus lisible grâce à une nouvelle instruction BASIC : l'instruction POUR (ou boucle POUR).

Le problème ci-dessus s'énonce de manière équivalente : pour les valeurs de I allant de 1 à 10, imprimer I et sa racine carrée. L'instruction POUR traduit cette phrase presque mot à mot de la manière suivante :

```
10  FØR I = 1 TØ 10
20  PRINT I, SQR (I)
30  NEXT I
40  END
```

La ligne 30 indique à l'ordinateur qu'il doit prendre la valeur suivante de I pour exécuter de nouveau la ligne 20.

Cette écriture est strictement équivalente à celle du Programme 7. La variable I est appelée *variable de contrôle* de la boucle POUR.

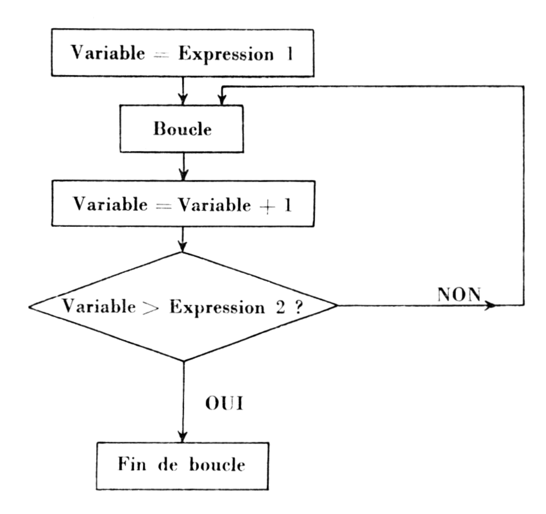
Il est possible de prendre des expressions comme valeurs initiale et finale de la variable de contrôle.

Une première forme générale de la boucle POUR est donc :

```
n° de ligne       FØR variable = expression 1 TØ expression 2
numéros suivants  boucle
numéro de ligne   NEXT variable
```

« numéros suivants boucle » schématise la succession des lignes dont les numéros sont compris entre celui de la ligne commençant par FØR et celui de la ligne où NEXT figure pour la première fois avec la même variable. Pour l'ordinateur, cette

écriture correspond exactement à l'organigramme suivant :



Il est souvent utile que la variable de contrôle progresse d'une valeur non nécessairement égale à 1, à chaque exécution de la boucle. Par exemple, si l'on ne veut imprimer que les nombres pairs et leurs racines carrées, il faut donner à I les valeurs 2, puis 4, puis 6...

On dit alors que les valeurs de I progressent par *pas* de 2, et on écrira :

```
10  FØR I = 2 TØ 10 STEP 2
20  PRINT I, SQR (I)
30  NEXT I
40  END
```

Le *pas* peut être une expression quelconque et l'écriture la plus générale d'une boucle POUR est la suivante :

```
de ligne           FØR variable=expression 1 TØ expression 2 STEP expression 3
numéros suivants   boucle
numéro de ligne    NEXT variable
```

Dans la première ligne de l'instruction POUR, STEP et l'expression qui le suit peuvent être omis : dans ce cas, le pas est égal à 1.

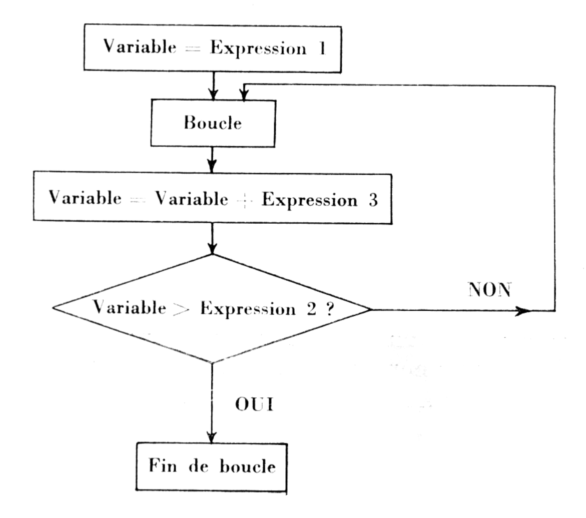
Si le pas n'est pas omis, sa valeur, c'est-à-dire celle de l'expression 3, doit être non nulle, sinon la boucle est exécutée indéfiniment.

La valeur du pas peut être positive ou négative.

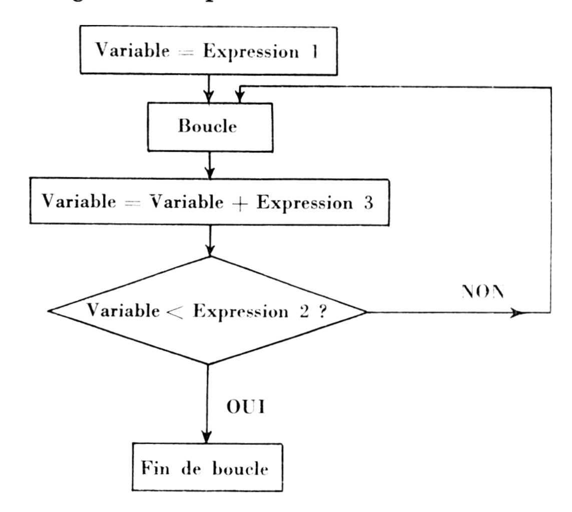
— Si elle est positive, il faut que la valeur de l'expression 2 soit supérieure à celle de l'expression 1, sinon la boucle ne sera exécutée qu'une seule fois (la variable de contrôle prenant comme seule valeur celle de l'expression 1) ;
— Si elle est négative, il faut, pour la même raison, que la valeur de l'expression 2 soit inférieure à celle de l'expression 1.

L'écriture la plus générale correspond aux deux organigrammes suivants, selon que la valeur du pas est positive ou négative :

1) Valeur positive du pas :



2) Valeur négative du pas :



Remarquons dans ce second cas, que la valeur de la variable de contrôle diminue chaque fois que la boucle est exécutée.

Exemple : Imprimer les nombres entiers de 10 à 1, et leurs racines carrées.

```
10  FØR I = 10 TØ 1 STEP — 1
20  PRINT I, SQR (I)
30  NEXT I
40  END
```

Remarque : Il est interdit de faire un branchement (par une instruction GØTØ ou une instruction conditionnelle) à partir d'une ligne extérieure à une boucle POUR vers une ligne intérieure à la boucle POUR (comprise entre les lignes commençant par FØR et NEXT). En effet, dans ce cas, au moment de l'exécution de la ligne intérieure à la boucle POUR, la variable de contrôle de cette boucle n'a pas reçu de valeur, puisque l'exécution de la ligne FØR correspondante n'a pas été faite.

Exemple : Le programme suivant est incorrect.

```
10  READ N
20  IF N = 1  THEN 40
30  FØR I = 1 TØ N
40  PRINT I, SQR (I)
50  NEXT I
55  DATA 5
60  END
```

A la ligne 20, il y a un branchement à la ligne 40 (lorsque N est égal à 1), c'est-à-dire à l'intérieur de la boucle POUR décrite aux lignes 30, 40 et 50. Donc, si N est égal à 1, la variable de contrôle I n'aura pas reçu normalement de valeur au moment de l'exécution de la ligne 40.

Par contre, il est possible d'effectuer un branchement à partir d'une ligne intérieure à une boucle POUR vers une autre ligne, intérieure ou non à cette boucle.

Exemples :

1) Imprimer les dix premiers nombres entiers et leurs racines carrées, sauf pour le nombre N (paramètre). Pour celui-ci, imprimer le double de sa racine carrée.

```
10  READ N
20  FØR I = 1 TØ 10
30  IF I = N  THEN 60
40  PRINT I, SQR (I)
50  GØTØ 70
60  PRINT 2 * SQR (I)
70  NEXT I
75  DATA 5
80  END
```

Remarquons que lorsque I est différent de N, on imprime I et sa racine carrée, puis on se branche à la ligne 70 (et non à la ligne 30 ou à la ligne 20) qui fait progresser I et recommencer, le cas échéant, la boucle.

2) Une liste de dix nombres étant donnée, déterminer si elle contient ou non le chiffre 0.

```
10  FØR I = 1 TØ 10
20  READ X
30  IF X = 0  THEN 70
40  NEXT I
50  PRINT "LA LISTE NE CØNTIENT PAS DE ZERØ"
60  STØP
70  PRINT " LA LISTE CØNTIENT UN ZERØ "
80  END
```

La boucle POUR des lignes 10 à 40 lit les nombres de la liste un à un et compare à chaque fois le nombre lu (variable X) à 0. En cas d'égalité, il y a branchement hors de la boucle à la ligne 70. S'il n'y a jamais égalité, la boucle POUR est exécutée pour les dix valeurs de la variable de contrôle et c'est ensuite la ligne 50 qui est exécutée.

VII.5. Boucles imbriquées

Les instructions à l'intérieur d'une boucle POUR peuvent être quelconques. En particulier, certaines d'entre elles peuvent décrire une autre boucle POUR : on dit alors que les deux boucles POUR sont *imbriquées*.

Exemple : Etant donné trois listes différentes de cinq nombres chacune, chercher et imprimer le nombre maximal de chaque liste.

Pour chaque liste, on lit d'abord le premier nombre X de la liste. Puis on lit successivement les autres nombres et on les compare à chaque fois à X. Si le nombre Y qui vient d'être lu est supérieur à X, on donne à X cette valeur Y. A tout moment, X représente donc le plus grand des nombres déjà lus dans une liste.

Le nombre de listes déjà examinées est représenté par un compteur I, variable de contrôle d'une première boucle POUR.

Le nombre de valeurs déjà examinées dans une liste est représenté par un compteur J, variable de contrôle d'une seconde boucle POUR.

```
 10  FØR I = 1 TØ 3
 20  READ X
 30  FØR J = 2 TØ 5
 40  READ Y
 50  IF Y < = X THEN 70
 60  LET X = Y
 70  NEXT J
 80  PRINT " LISTE " I; " NØMBRE MAXIMUM = " X
 90  NEXT I
100  END
```

Pour que l'ordinateur exécute correctement deux boucles POUR imbriquées, il convient de respecter certaines règles d'écriture.

Règle 1 : Les variables de contrôle des boucles POUR imbriquées doivent être différentes.

C'est pour cela que dans l'exemple précédent nous avons pris J comme variable de contrôle de la boucle POUR des lignes 30 à 70, et I comme variable de contrôle de l'autre boucle.

Règle 2 : Les boucles POUR doivent être effectivement imbriquées, c'est-à-dire qu'à l'intérieur d'une boucle POUR, il ne peut pas apparaître une ligne FØR sans la ligne NEXT correspondante, ni une ligne NEXT sans la ligne FØR correspondante.

Exemple : Le programme suivant est incorrect :

```
10  FØR I = 1 TØ 3
20  READ X
30  FØR J = 2 TØ 5
40  READ Y
50  IF Y < = X  THEN 90
60  LET X = Y
70  NEXT I
```

```
 80  PRINT " LISTE " I; " NØMBRE MAXIMUM = " X
 90  NEXT J
100  END
```

Bien entendu, le nombre de boucles imbriquées n'est pas limité à deux et, d'autre part, à l'intérieur d'une boucle POUR peuvent figurer deux boucles POUR non imbriquées (et pouvant avoir alors la même variable de contrôle).

EXERCICES

— Quel est le processus suivi par l'ordinateur pour l'évaluation de la condition :
B ✕ B — 4 ✕ A ✕ C > LØG (1)

— L'instruction
20 IF A < > 0 THEN GØ TØ 5
est-elle correcte ?

— Quelles sont les formes de la boucle PØUR ? Utilité de cette instruction ?

— Ecrire à l'aide d'un compteur et de l'instruction conditionnelle un programme équivalent à la séquence d'instructions suivantes :

```
25  FØR I = 10 TØ 0 STEP — 2
30  READ A
35  LET B (I) = A
40  READ A
45  LET B (I — 1) = A
50  NEXT I
10  DATA 100, 80, 60, 40, 20
60  END
```

— Calculer la somme des N premiers nombres pairs.
Application : N = 50, N = 100.

— Pouvez-vous sortir d'une boucle PØUR ? A partir du « corps » d'une boucle PØUR, peut-on se brancher à la première instruction d'une autre boucle PØUR ? Et à l'intérieur de cette autre boucle PØUR ? Justifier votre dernière réponse.

— En cas de branchement hors de la boucle PØUR, que devient la valeur de la variable ? En fin d'exécution d'une boucle PØUR, quelle est la valeur de la variable ?

— Soient M1, le revenu de M. Dupont, M2 le revenu de Mme Dupont.

Si chacun de ces revenus est inférieur à 5 000 F, le ménage ne paie pas d'impôt. Dans le cas contraire,

Calculer le revenu global du ménage.

Si ce revenu global est inférieur à 30 000 F, le calcul simple de l'impôt se fera comme suit :

— imposition à 5 % pour la tranche comprise entre 5 000 et 10 000 F ;
— imposition à 10 % pour la tranche comprise entre 10 000 et 20 000 F ;
— imposition à 15 % pour la tranche comprise entre 20 000 et 30 000 F.

Dresser l'organigramme et écrire le programme permettant de savoir si le ménage paie ou non des impôts et combien. Quelle instruction de lecture des données avez-vous intérêt à utiliser ?

Chapitre VIII

ENTRÉES DES DONNÉES A PARTIR DU TERMINAL VARIABLES ALPHANUMÉRIQUES

Au chapitre V, nous avons indiqué comment fournir des données à un programme : description des données par l'instruction DATA et leur lecture par l'instruction READ.

Pour un utilisateur qui veut exécuter le programme avec ses données, ces instructions sont peu agréables d'emploi, car elles obligent cet utilisateur à connaître le programme de manière relativement détaillée. En effet, il doit connaître :

— l'ordre dans lequel sont lues les données et les variables auxquelles elles sont affectées pour qu'il puisse fournir ses propres données dans un ordre cohérent avec le programme tel qu'il a été écrit ;

— les numéros des lignes DATA à modifier (pour introduire ses données) avant exécution du programme.

Cette façon de faire représente une charge de travail importante pour l'utilisateur lorsque le programme est complexe, et elle est, de toute manière exclue pour un utilisateur qui ignore et veut continuer d'ignorer la programmation. Un tel utilisateur ne peut pas analyser et programmer ses propres problèmes, mais il veut pouvoir utiliser des programmes « tout faits » (que les Anglo-Saxons appellent des *package*), en ignorant complètement la manière dont ces programmes ont été écrits.

Il est donc indispensable de disposer dans le langage d'une

instruction qui permette de lire des données sans pour autant modifier le programme.

La solution la plus simple est que cette instruction pose une question à l'utilisateur *sur le terminal et au moment de l'exécution du programme.* En réponse l'utilisateur fournira sa donnée (un nombre par exemple) et le programme s'exécutera avec cette donnée (ou plusieurs données qui feront l'objet de plusieurs questions).

Dans ce chapitre, nous montrons que cette possibilité existe en langage BASIC.

Nous verrons aussi que les réponses données par l'utilisateur peuvent être des nombres et aussi des messages en « langage clair ». Ce sont des chaînes dont nous avons déjà parlé au chapitre IV. Ces chaînes constituent un autre type d'objets qu'un programme peut manipuler, de la même façon que les nombres ou les variables simples. Enfin, nous donnerons les différentes façons d'utiliser une chaîne en langage BASIC.

VIII.1. Instruction INPUT

Le langage BASIC dispose d'une instruction de lecture de données qui diffère de l'instruction READ, car elle ne nécessite pas la description des données dans les lignes DATA du programme.

La forme la plus simple de cette instruction est la suivante :

INPUT variable

Exemple :

```
10 INPUT N
```

Lorsque l'ordinateur exécute une telle instruction, il imprime un point d'interrogation sur le terminal, *à l'endroit où se trouve à ce moment la tête d'écriture* et attend que l'utilisateur frappe une succession de caractères. Cette succession de caractères doit être suivie d'un « retour chariot ». Dès la frappe de ce dernier, l'ordinateur reprend le contrôle. Il examine alors la succession de caractères qui ont été frappés. Si la variable qui suit le

symbole INPUT dans l'instruction qu'il exécute est une variable simple, la succession de caractères doit représenter un nombre BASIC correct. Si c'est le cas, l'ordinateur affecte ce nombre à la variable qui suit le symbole INPUT.

Si ce n'est pas le cas, l'ordinateur imprimera un message d'erreur sur le terminal, puis un nouveau point d'interrogation à la ligne suivante et attendra l'entrée d'une nouvelle succession de caractères qu'il vérifiera avant affectation.

Exemple : Si, lors de l'exécution de l'instruction INPUT N, l'utilisateur frappe 452 en réponse au point d'interrogation imprimé par l'ordinateur, la variable N aura la valeur 452 après exécution de la dite instruction.

Un simple point d'interrogation n'est pas en soi une question très explicite. En particulier, il ne permet pas à l'utilisateur de savoir quel genre d'informations l'ordinateur attend de lui.

Il est possible de remédier à cet inconvénient en combinant les instructions PRINT et INPUT. En effet, il suffit de faire précéder immédiatement une instruction INPUT d'une instruction PRINT imprimant un message, c'est-à-dire une chaîne.

Exemple :

```
10  PRINT "VALEUR DE A";
20  INPUT A
```

L'ordinateur exécute d'abord la ligne 10 et imprime donc le message :

VALEUR DE A

Puis, il exécute l'instruction 20 ; il imprime donc un point d'interrogation à l'endroit où se trouve actuellement la tête d'écriture, c'est-à-dire après le message VALEUR DE A, car nous avons terminé la ligne 10 par un point-virgule (voir chap. IV).

L'utilisateur verra donc apparaître sur le terminal la question :

VALEUR DE A ?

et il donnera cette valeur, en face de la question, sur la même ligne.

Remarquons que si nous n'avions pas terminé la ligne 10 par un point-virgule, le point d'interrogation aurait été frappé au début de la ligne suivant le message.

De même que dans l'instruction READ, il est possible d'entrer plusieurs données à la fois à partir du terminal.

La forme la plus générale de l'instruction INPUT est la suivante :

INPUT liste de variables.

Les variables de la liste doivent être séparées par des virgules. Dans ce cas, on peut taper après le point d'interrogation imprimé par l'ordinateur, autant de valeurs qu'il y a de variables dans la liste. Toutes ces valeurs, lorsque ce sont des nombres, doivent être séparées par des virgules ou des espaces.

Exemple :

10 INPUT X, Y, Z

Après le point d'interrogation correspondant à cette instruction, l'utilisateur peut taper :

— 10 . 52 ␣ 37, 10 E — 5

L'ordinateur affecte les valeurs tapées aux variables de la liste : la première valeur est affectée à la première variable, la seconde à la seconde variable, etc. Dans l'exemple précédent, après exécution de la ligne 10, les valeurs de X, Y et Z sont respectivement : — 10.52, 37 et 10 E — 5.

Si l'utilisateur fournit moins de valeurs qu'il n'y a de variables dans la liste, l'ordinateur affecte les valeurs tapées aux variables correspondantes et réimprime un point d'interrogation pour demander les valeurs à affecter aux autres variables et ceci autant de fois qu'il faut pour affecter toutes les variables de la liste.

Si, au contraire, l'utilisateur frappe plus de valeurs qu'il n'y a de variables dans la liste, les valeurs en excédent sont ignorées... sauf, éventuellement, si une autre instruction INPUT est exécutée assez tôt après pour que l'ordinateur considère que les valeurs tapées en excédent la première fois, sont celles à

prendre en compte dans cette nouvelle instruction INPUT. Nous déconseillons vivement une telle utilisation car son effet est très aléatoire.

Application : Nous pouvons réécrire le Programme 6 du chapitre VII, c'est-à-dire la résolution complète d'une équation du second degré, en utilisant cette nouvelle possibilité :

Programme 8 :

```
 10 PRINT " NØMBRE D'EQUATIØNS ";
 20 INPUT N
 30 LET I = 1
 40 PRINT " EQUATIØN " I
 50 PRINT " VALEUR DE A ";
 60 INPUT A
 70 PRINT " VALEUR DE B ";
 80 INPUT B
 90 PRINT " VALEUR DE C ";
100 INPUT C
105 LET D = B ↑ 2 — 4 * A * C
110 IF A < > 0  THEN 200
120 IF B  =  0  THEN 150
130 PRINT " 1ER DEGRE : RACINE = " — C/B
140 GØTØ 290
150 IF C = 0  THEN 180
160 PRINT " IMPØSSIBLE "
170 GØTØ 290
180 PRINT " INDETERMINE "
190 GØTØ 290
200 IF D < 0  THEN 280
210 IF D = 0  THEN 260
220 LET X1 = (— B + SQR(D)) / (2 * A)
230 LET X2 = (— B — SQR(D)) / (2 * A)
240 PRINT " 1RE RACINE = " X1 ; " 2DE RACINE = " X2
250 GØTØ 290
260 PRINT " 2D DEGRE : RACINE UNIQUE = " — B / (2 * A)
270 GØTØ 290
280 PRINT " PAS DE RACINES "
```

```
290  I = I + 1
300  IF I < = N   THEN 40
310  END
```

Une exécution de ce programme donnera par exemple les résultats suivants :

```
NØMBRE D'EQUATIØNS ? 3
EQUATIØN 1
VALEUR DE A ? 2
VALEUR DE B ? 7.68
VALEUR DE C ? — 5.94
1ERE RACINE = 4.5
2DE RACINE = 0.66
EQUATIØN 2
VALEUR DE A ? 0
VALEUR DE B ? — 5
VALEUR DE C ? 6.25
1ER DEGRE : RACINE UNIQUE = 2.5
EQUATIØN 3
VALEUR DE A ? 8.25
VALEUR DE B ? — 2.1
VALEUR DE C ? 4
PAS DE RACINES
```

Remarquons que cette écriture, contrairement à celle du Programme 6, représente un algorithme complet puisqu'elle est totalement indépendante des valeurs données à N, A, B et C et qu'elle n'a pas besoin d'être modifiée pour fournir les résultats dans chaque cas particulier.

VIII.2. Variables alphanumériques

Le Programme 8 peut encore présenter une certaine gêne pour l'utilisateur. En effet, l'utilisateur est obligé de connaître, au moment de l'exécution, le nombre d'équations à résoudre, et une fois ce nombre fourni, il ne peut plus le modifier.

Si, par exemple, il veut résoudre deux équations de plus

qu'il n'avait prévu, il sera obligé de faire exécuter une nouvelle fois le programme en donnant à N la valeur 2.

Il serait bien sûr plus agréable d'indiquer, après la résolution d'une équation si l'utilisateur veut ou non résoudre une autre équation. Cela se fait en modifiant comme suit le Programme 8 :

— suppression des lignes 10 et 20
— remplacement de la ligne 300 par :

```
300  PRINT " AUTRE EQUATIØN " ;
302  INPUT R
```

Après avoir résolu une équation, l'ordinateur posera la question :

AUTRE EQUATIØN ?

et attendra la réponse de l'utilisateur (ligne 302).

Cette réponse sera affectée à la variable R. Ce doit donc être un nombre et on peut adopter par exemple comme convention que ce nombre est 0, si la réponse est NON et 1 si la réponse est OUI.

Il suffit donc pour accéder au désir de l'utilisateur de tester après la ligne 300, si la valeur de R est 0 ou 1 et selon le cas s'arrêter ou recommencer la résolution d'une autre équation. Nous écrirons donc :

```
305  IF R = 1  THEN 40
```

Cette façon de faire n'est pas encore idéale, car la réponse ne peut pas être donnée en « clair », par le mot OUI ou le mot NON.

L'introduction d'une réponse en clair est rendue possible en langage BASIC par un nouveau type d'objets que le langage peut manipuler : *les variables alphanumériques.*

Comme une variable simple, une variable alphanumérique représente le nom d'une case dans la mémoire de l'ordinateur, mais cette case peut contenir une succession (1) de caractères quelconques, c'est-à-dire une chaîne et non plus simplement un nombre.

(1) En général, le nombre de caractères de cette succession est limité. Cette limite dépend du système de temps partagé.

Pour différencier les variables alphanumériques des variables simples, on les écrit au moyen d'une lettre suivie du symbole $.

Exemples :

R$, A$

Une variable alphanumérique représente une « valeur », qui est une chaîne et qui peut être modifiée comme une variable simple.

En particulier, il est possible d'entrer une chaîne à partir du terminal et de l'affecter à une variable alphanumérique.

Dans l'exemple précédent, on pourra écrire à la place de la ligne 302 :

```
302  INPUT R$
```

Dans ce cas, l'ordinateur affectera à la variable R$ la chaîne constituée par tous les caractères (y compris les espaces) tapés par l'utilisateur après le point d'interrogation et avant le retour - chariot, pourvu que le nombre de caractères tapés ne dépasse pas la limite du nombre acceptable de caractères pour une variable alphanumérique.

VIII.3. Calculs sur les variables alphanumériques

Il est aussi possible de tester la valeur d'une variable alphanumérique, c'est-à-dire par exemple de savoir si une variable alphanumérique est égale à une chaîne donnée écrite explicitement dans le programme.

Exemple : Dans le Programme 8, modifié comme il a été indiqué ci-dessus, on peut tester si la réponse de l'utilisateur est positive, c'est-à-dire si la valeur de R$ est la chaîne OUI, au moyen de l'instruction conditionnelle suivante :

```
305  IF R$ = " OUI "  THEN 40
```

A la forme générale des conditions que nous avons données au chapitre VII, § VII.1, il convient donc d'ajouter les trois formes suivantes :

— variable de chaîne opérateur de relation variable de chaîne ;
— variable de chaîne opérateur de relation chaîne ;
— chaîne opérateur de relation variable de chaîne.

Rappelons que la chaîne doit être entourée de guillemets dans l'écriture du programme.

Dans le cas où l'opérateur de relation utilisé est = ou < > (différent), la signification de la condition est évidente.

Pour tous les autres opérateurs de relation, la condition est examinée en tenant compte des codes internes à l'ordinateur, de chacun des caractères des chaînes à comparer. Nous n'insisterons pas sur ces cas qui dépendent beaucoup du système de temps partagé.

Les variables alphanumériques peuvent être aussi lues à partir d'une zone de données.

Dans ce cas, la ligne DATA doit contenir comme valeur une chaîne.

Exemple :

```
50  DATA " IMPØSSIBLE "
```

L'instruction READ qui lira cette donnée doit alors comporter une variable alphanumérique.

D'autre part, on peut mélanger des nombres et des chaînes dans une instruction DATA, ainsi que des variables simples et des variables alphanumériques dans une instruction READ. Cependant, *lorsque la variable à lire est simple, le pointeur de la zone de données doit repérer un nombre et lorsqu'elle est alphanumérique, ce pointeur doit repérer une chaîne.* En cas de défaut de cette règle, l'ordinateur imprimera un message d'erreur.

Le fonctionnement des instructions DATA et READ reste le même que celui que nous avons indiqué au chapitre V.

Exemple :

```
1  DATA — 1,“ IMPØSSIBLE ”,
   2.5, “ PAS ⊔ DE ⊔ RACINES ”
2  READ X, A$, Y, B$
```

Après exécution de la ligne 2, les valeurs de X et Y sont respectivement — 1 et 2.5 et celles de A$ et B$ sont respectivement les chaînes : IMPØSSIBLE et PAS ⊔ DE ⊔ RACINES.

Il est possible aussi d'affecter à une variable alphanumérique une chaîne ou une autre variable alphanumérique.

Nous compléterons donc l'écriture de l'instruction d'affectation (chap. III, § III.1) par les deux formes suivantes :

LET variable alphanumérique = chaîne
LET variable alphanumérique = variable alphanumérique.

Après une instruction d'affectation de ce type, la variable alphanumérique à gauche du signe = prend la valeur représentée par la chaîne ou la variable alphanumérique à droite du signe =.

Exemple :

```
10  LET X$ = “ NON ”
20  LET Y$ = X$
```

Après l'exécution de ces deux lignes, les variables X$ et Y$ représentent toutes deux la chaîne NON.

Enfin, il est possible d'imprimer la valeur d'une variable alphanumérique.

Exemple :

```
10  LET X$ = “ NON ”
20  PRINT X$
30  LET X$ = “ OUI ”
40  PRINT X$
50  END
```

Ce programme imprime le message NON puis le message OUI à la ligne suivante.

Il est bien sûr possible de mélanger dans une instruction PRINT des variables alphanumériques, des chaînes, des expressions ou des tabulations. Le fonctionnement de l'instruction PRINT reste le même que celui que nous avons décrit au chapitre IV.

EXERCICES

— Quelles différences faites-vous dans l'utilisation des deux instructions de lecture des données :
READ et INPUT

Quels sont leurs effets respectifs ? Quelle est la plus souple ? La mieux adaptée à l'utilisation d'un même programme pour la résolution de problèmes où les données évoluent ?

— Comment l'ordinateur ou vous-même reconnaissez un identificateur de variable alphanumérique ?

— La lecture d'une variable alphanumérique peut-elle se faire indifféremment par l'un ou l'autre des ordres READ, INPUT ?

— Si, par inattention, vous affectez un nombre à un identificateur de chaîne, est-ce une erreur ?

— Pouvez-vous affecter une variable alphanumérique à une autre variable alphanumérique ?

— Pouvez-vous utiliser les opérateurs de relation présentés au chapitre VII avec les variables alphanumériques ?

Chapitre IX

LES LISTES ET LES TABLES EN « BASIC »

Dans les chapitres qui précèdent, une variable, simple ou alphanumérique, avait pour but de représenter une seule valeur : un nombre ou une chaîne.

Ceci est une restriction qui présente un inconvénient dans le cas où un grand nombre de valeurs variables et différentes se rapportent en fait à la même entité. Par exemple, dans un programme qui effectue des calculs sur les quantités vendues d'un article pendant cinquante mois, il faudrait utiliser cinquante variables simples différentes, chacune d'elles représentant la quantité vendue dans un mois.

Cette façon de faire présente deux inconvénients :

— le nombre de variables simples nécessaires à un programme risque d'être très grand ; or, ce nombre est limité à 286, compte tenu des possibilités d'écriture de variables simples différentes (voir chap. II) ;
— le plus souvent, les calculs sur des variables se rapportant à une même entité seront semblables ; or, si ces variables sont différentes, il faudra effectuer pratiquement le même calcul autant de fois qu'il y a de variables différentes.

Exemple : Si les variables V1, V2, ..., V9 sont les quantités vendues d'un article respectivement pendant les mois de janvier, février, ..., septembre, et si X1, X2, ..., X9 représentent des informations analogues pour un autre article, le calcul de la somme des quantités vendues par mois pour les deux articles et l'affectation de ces sommes à des variables S1, S2, ..., S9, obligera à écrire les neuf instructions d'affectation suivantes :

```
10  LET S1 = V1 + X1
20  LET S2 = V2 + X2
```

```
30 LET S3 = V3 + X3
 .
 .
 .
90 LET S9 = V9 + X9
```

Cette écriture serait considérablement simplifiée si nous pouvions représenter les valeurs de V1 à V9 par un seul nom. De même pour les valeurs de X1 à X9 et de S1 à S9.

Ce nom qui serait celui d'une variable spéciale, désignerait alors non plus une seule valeur mais plusieurs valeurs rangées dans un certain ordre.

Nous allons voir dans ce chapitre que cela est possible en langage BASIC et nous introduirons un nouveau type d'objets que le langage BASIC peut manipuler : les variables de liste et les variables de tableau.

Nous verrons en particulier que la description de ce type d'objets nécessite quelques précautions d'écriture dans un programme, puis nous étudierons la façon d'utiliser chacune des valeurs représentées par une variable de tableau. Enfin, nous verrons que les valeurs représentées par une variable de tableau peuvent être manipulées « en bloc ».

IX.1. Description des variables de liste

Comme nous l'avons dit dans l'introduction de ce chapitre, il est possible de représenter un ensemble de plusieurs valeurs au moyen d'une seule variable.

Par exemple, la variable V représentera le détail des quantités vendues d'un article pour les mois de janvier à septembre, c'est-à-dire neuf valeurs différentes.

Ceci revient à dire que la variable V représente non pas le nom d'une seule case, mais de neuf cases successives, en mémoire de l'ordinateur.

Ces cases sont rangées dans un certain ordre et numérotées de un en un à partir de 1.

La quantité vendue en janvier se trouve dans la case 1, celle vendue en février dans la case 2, etc.

Le contenu de chaque case peut varier comme nous le verrons plus loin.

En fait, une telle variable représente une liste de valeurs et nous l'appellerons *variable de liste*. Son écriture est constituée d'une seule lettre.

Comme le nombre de valeurs représentées par cette variable peut être grand, l'ordinateur doit connaître ce nombre avant l'exécution du programme afin de réserver en mémoire la place nécessaire à toutes ces valeurs.

Cette réservation, appelée *dimensionnement*, se fait au moyen d'une instruction de la forme suivante :

DIM variable de liste (nombre)

le nombre entre parenthèses doit être un nombre entier positif explicitement écrit. Il représente le nombre de cases à réserver pour la liste dont le nom est celui de la variable de liste indiquée.

Remarques :

1. En fait, les cases correspondant à une variable de liste sont numérotées à partir de 0 (il existe une case numérotée 0 qui peut contenir une valeur).

Ainsi, une instruction de la forme :

```
10  DIM  A(8)
```

réserve *neuf* cases numérotées de 0 à 8 et non pas huit.

2. Il est possible de donner le même nom à une variable simple et à une variable de liste.

Exemple :

```
10  DIM  A (8)
20  LET  A = 1
 .
 .
 .
```

A la ligne 20, la variable A est une variable simple, représentant une seule valeur et non pas la variable de liste dimen-

sionnée à la ligne 10. Nous verrons plus loin qu'il n'y a aucune confusion possible entre les deux types de variables.

Il n'est pas toujours nécessaire de dimensionner une variable de liste. Lorsqu'on utilise une variable de liste ayant moins de onze éléments (voir plus loin), on n'est pas obligé de la dimensionner. L'ordinateur réservera automatiquement onze cases numérotées de 0 à 10 pour cette variable.

Cependant, si on n'utilise pas chacune de ces onze cases, on aura intérêt à dimensionner la variable au nombre exact de ses éléments pour éviter une perte inutile de place en mémoire de l'ordinateur.

Exemple : Si un programmeur utilise les cases numérotées de 0 à 4 d'une variable de liste X, il a intérêt à écrire :

```
10   DIM X(4)
```

sinon l'ordinateur lui réservera aussi les cases 5 à 10 qui, bien qu'inutilisées prendront de la place en mémoire.

Le dimensionnement réserve une place fixe pour une variable de liste. Il doit précéder dans le programme toutes les instructions qui utilisent cette variable de liste (voir plus loin). Sur certains systèmes de temps partagé, cette restriction est supprimée. Le dimensionnement peut réserver une place variable, c'est-à-dire que dans une instruction DIM, la variable de liste peut être suivie d'une *expression quelconque* entre parenthèses. La valeur de cette expression est calculée au moment de l'exécution de l'instruction DIM et elle détermine la place à réserver pour la variable de liste correspondante.

En outre, une variable de liste peut être redimensionnée en cours de programme par une nouvelle instruction DIM, qui sera donc placée à n'importe quel endroit d'un programme.

Exemple :

```
10   INPUT N
20   DIM A(N)
 .
 .
 .
```

```
50 DIM A(N + 1)
.
.
.
```

Il est possible aussi d'utiliser des variables de liste dont les éléments sont des chaînes et non plus des nombres. Chacune des variables alphanumériques de liste s'écrit au moyen d'une lettre suivie du symbole $.

Elles doivent faire l'objet d'un dimensionnement régi par les mêmes règles que celles d'une variable de liste.

Exemple :

```
10 DIM A$ (5)
```

Cette instruction réserve la place pour une liste de six chaînes numérotées de 0 à 5. Cette liste est appelée A$.

Enfin, il est possible de dimensionner plusieurs variables de liste ou variables alphanumériques de liste, mélangées dans une seule instruction DIM, dont la forme la plus générale est donc :

DIM liste de dimensionnement

chaque dimensionnement est constitué d'une variable de liste ou d'une variable alphanumérique de liste suivie d'un nombre entier positif entre parenthèses.

Les dimensionnements sont séparés par des virgules.

Exemple :

```
10 DIM A(5), B(30), A$(2), C(12)
```

IX.2. Utilisation d'une variable de liste

Il est possible d'utiliser individuellement chacune des valeurs d'une liste. Nous avons dit qu'une variable de liste représentait globalement une succession de cases dans la mémoire. On peut désigner explicitement une de ces cases en faisant suivre la

variable de liste du numéro d'ordre de cette case, mis entre parenthèses.

Exemple : V(0) représente la case de numéro 0 de la liste V, V(1) la case de numéro 1, etc.

Cette écriture est appelée une *variable indicée* : elle représente une seule valeur comme une variable simple, mais cette valeur dépend de l'*indice*, c'est-à-dire du numéro d'ordre placé entre parenthèses.

Exemple : V(1) et V(4) ne représentent pas la même case, donc n'ont pas nécessairement la même valeur. Mais ces deux cases appartiennent toutes deux à celles de la variable de liste V.

En fait, l'intérêt des variables de liste serait faible, si les seuls indices qu'on pouvait écrire dans une variable indicée étaient des nombres.

C'est pourquoi, l'écriture générale d'une variable indicée est la suivante :

variable de liste (expression).

Lorsque l'ordinateur rencontre dans un programme une telle variable indicée, il calcule la valeur *n* de l'expression entre parenthèses. Si cette valeur n'est pas entière, il l'arrondit à l'entier le plus proche. Si elle n'est pas positive, l'ordinateur signale l'erreur.

La variable indicée représente pour l'ordinateur la ($n + 1$ ième) case de la liste correspondante (rappelons que le numérotage des cases d'une liste commence à 0).

Exemple : Si la valeur de I est 2 et celle de J est 4, la variable indicée

V (I + 2 * J)

représente la onzième case de la liste V, celle correspondant à V(10).

Encore faut-il que dans la liste considérée, il existe une case de numéro d'ordre *n*, c'est-à-dire que cette liste contienne au moins $n + 1$ cases (numérotées de 0 à n), sinon l'ordinateur imprime un message d'erreur sur le terminal.

Exemple : Si, dans l'exemple précédent, la variable V a été dimensionnée par :

```
10  DIM V(5)
```

l'ordinateur imprimera un message d'erreur lorsqu'il rencontrera V (I + 2 $\times$ J), car V ne contient que six cases.

Il faut donc prendre garde à ne pas utiliser de variables indicées dont l'indice dépassera la valeur du dimensionnement de la liste correspondante (ou 10 si ce dimensionnement a été omis).

Une variable indicée représente donc une seule valeur et peut, par conséquent, être utilisée exactement comme une variable simple.

En particulier, elle peut apparaître dans une expression ; on pourra donc imprimer la valeur d'une variable indicée ; elle peut aussi être lue par une instruction READ ou une instruction INPUT ; elle peut également apparaître à gauche du signe = dans une affectation, c'est-à-dire que sa valeur peut être modifiée. Enfin, comme elle peut être utilisée dans une expression, elle peut faire l'objet de test.

D'une manière plus précise, partout où, dans les chapitres précédents, nous avons parlé de variable, il faut entendre par ce terme, variable simple ou variable indicée.

Exemples :

1)

```
10  DIM V(2)
20  INPUT V(0)
30  LET V(1) = V(0) ↑ 2
40  LET V(2) = V(0) + V(1)
50  PRINT V(0), V(1), V(2)
60  END
```

Une liste de trois éléments V(0), V(1) et V(2) est dimensionnée à la ligne 10.

Le premier élément V(0) de la liste est lu à partir du terminal à la ligne 20.

A la ligne 30, on affecte comme valeur au second élément,

le carré du premier et à la ligne 40, comme valeur du troisième élément, la somme des deux premiers.

Enfin, on imprime les trois éléments à la ligne 50.

2) La boucle POUR apparaît d'un emploi particulièrement agréable dans la manipulation des listes.

```
 10  DIM V(12), X(12), S(12)
 20  FØR I = 1 TØ 12
 30  INPUT V(I)
 40  INPUT X(I)
 50  NEXT I
 60  FØR I = 1 TØ 12
 70  LET S(I) = V(I) + X(I)
 80  PRINT S(I)
 90  NEXT I
100  END
```

A la ligne 10, on dimensionne trois listes qui représentent, par exemple, la première les quantités vendues d'un article pendant une année, la seconde les quantités vendues d'un autre article pendant la même période et la troisième la somme des quantités vendues pour les deux articles.

La première boucle POUR (lignes 20 à 40) permet d'entrer à partir du terminal les valeurs de V(1), X(1), puis V(2), X(2), etc.

La seconde boucle POUR (lignes 60 à 90) calcule mois par mois la somme des quantités vendues et place cette somme dans la liste S à la case correspondante (dans S(1), on a V(1) + X(1), dans S(2), on a V(2) + X(2), etc.). En outre, elle imprime sur le terminal ces différentes sommes.

3) Le programme suivant lit les éléments d'une liste L, à partir d'une ligne DATA et cherche le nombre maximal M de la liste.

```
 10  DIM L(10)
 20  FØR I = 0 TØ 10
 30  READ L(I)
 40  NEXT I
 50  LET M = L(0)
```

```
 60 FØR I = 1 TØ 10
 70 IF L(I) < = M  THEN 90
 80 LET M = L(I)
 90 NEXT I
100 PRINT M
110 DATA 7, 3, 2, 5.2, 12, 6, 5, 8, 1, 8.9
120 END
```

Remarque : Rien n'interdit à l'indice d'une variable indicée d'être une expression contenant elle-même des variables indicées.

Exemple :

```
10 LET V (I + X(J)) = 5
```

IX.3. Variables de tableau

Le langage BASIC ne se limite pas aux listes : il permet aussi de décrire et de manipuler des tableaux.

La notion de tableau est fréquemment utilisée dans la vie courante.

Exemple : Trois articles sont distribués par 5 magasins différents.

Les quantités vendues de chaque article par chaque magasin sont données dans le tableau suivant :

	Magasin				
Article	**1**	**2**	**3**	**4**	**5**
1	8	14	3	7	9
2	11	2	21	8	5
3	4	5	7	16	10

Le magasin 2 a vendu 5 articles du type 3, le magasin 5 a vendu 9 articles du type 1, etc. Ce tableau est constitué de 15 cases rangées en 3 lignes de 5 cases chacune. Il peut être représenté en BASIC par une seule variable identifiée par une lettre : une telle variable est appelée ***variable de tableau.***

Les variables de tableau utilisées dans un programme doivent être dimensionnées. Ce dimensionnement diffère de celui des variables de listes en ce sens qu'il fait apparaître, non plus le nombre de cases, mais le nombre de lignes et le nombre de colonnes du tableau.

L'instruction correspondante a la forme suivante :

DIM variable de tableau (nombre, nombre)

les deux nombres entre parenthèses doivent être des nombres entiers et positifs : le premier représente le nombre de lignes, le second le nombre de colonnes à réserver pour le tableau. En fait, les lignes et les colonnes sont numérotées à partir de 0.

Ainsi une instruction de la forme :

```
10   DIM A(3, 5)
```

réserve 4 lignes numérotées de 0 à 3 et 6 colonnes numérotées de 0 à 5, donc en tout 24 cases.

Remarques :

1) Une variable de liste et une variable de tableau ne peuvent pas avoir le même nom.

Exemple : l'écriture suivante est interdite :

```
10   DIM A(5)
20   DIM A(3, 5)
```

2) Il est possible de dimensionner plusieurs tableaux dans une même instruction DIM, et même d'y mélanger des dimensionnements de listes et de tableaux.

Exemple :

```
10   DIM A(5), A$(3), B(3, 5), C(2, 12), D(4)
```

Ainsi la forme la plus générale d'une instruction DIM est la suivante :

DIM liste de dimensionnements

chaque dimensionnement peut être un dimensionnement de

liste, de liste alphanumérique ou de tableau. Les différents dimensionnements de la liste sont séparés par des virgules.

3) Il n'est pas possible d'utiliser des tableaux de chaînes.

4) Le dimensionnement n'est pas nécessaire si le nombre de lignes et le nombre de colonnes utiles du tableau sont tous deux inférieurs à onze : l'ordinateur réserve automatiquement 11 lignes et 11 colonnes numérotées de 0 à 10, donc 121 cases pour chaque tableau utilisé sans être dimensionné.

Cependant, si toutes les colonnes ou toutes les lignes ne sont pas utilisées, il y a intérêt à dimensionner le tableau pour gagner de la place en mémoire.

Exemple : Si un programmeur utilise les lignes numérotées de 0 à 4 et les colonnes numérotées de 0 à 7 d'un tableau X, il a intérêt à écrire :

```
10   DIM X(4, 7)
```

sinon l'ordinateur réservera en plus les lignes 5 à 10 et les colonnes 8 à 10 qui seront inutilisées.

Il est évident que la taille possible pour les listes et les tableaux utilisés par un programme est limitée et dépend de la taille de la mémoire centrale et de la taille du programme d'utilisation de ces tableaux. Il convient donc d'éviter au maximum le gaspillage de place occupée par les listes et les tableaux.

Si l'un des deux nombres de ligne ou de colonne est supérieur à onze, le dimensionnement est obligatoire.

Exemple : Si le programme utilise un tableau T à 3 lignes et à 14 colonnes, il faudra écrire :

```
10   DIM T(2, 13)
```

5) Le dimensionnement réserve une place fixe pour un tableau, sauf dans le cas particulier de certains systèmes de temps partagé (voir le § IX.1). Il doit donc précéder dans le programme toutes les instructions qui utilisent la variable de tableau correspondante.

IX.4. Utilisation des variables de tableau

Comme pour les variables de liste, il est possible d'utiliser individuellement les valeurs d'un tableau. Pour cela, il suffit de faire suivre la variable de tableau du numéro d'ordre de la ligne et du numéro d'ordre de la colonne à l'intersection desquelles se trouve la case désirée.

Exemples :

T(0, 0) représente dans le tableau T, la case à l'intersection de la ligne 0 et de la colonne 0.

T(1, 0) représente la case à l'intersection de la ligne 1 et de la colonne 0, etc.

Ainsi, toutes les cases successives de la colonne 0 sont représentées par T(0, 0), T(1, 0), T(2, 0)...

Celles de la colonne 1 sont représentées par T(0, 1), T(1, 1), T(2, 1)... Inversement, les cases successives de la ligne 0 sont T(0, 0), T(0, 1), T(0, 2)...

Une telle écriture est encore appelée par extension *variable indicée.* Elle représente une seule valeur, qui dépend des deux indices écrits entre parenthèses.

L'écriture la plus générale d'une variable indicée de tableau est la suivante :

variable de tableau (expression 1, expression 2).

Lorsque l'ordinateur rencontre dans un programme une telle variable indicée, il calcule les valeurs n et m des expressions 1 et 2, comme dans le cas des listes.

La variable indicée est alors celle que l'ordinateur trouve dans la case du tableau correspondant, située à l'intersection de la $(n + 1)$-ième ligne et de la $(m + 1)$-ième colonne.

Il faut bien remarquer que le premier indice sélectionne la ligne et le second la colonne.

Exemple : Si la valeur de I est 2 et celle de J est 3, la variable indicée T(I, J) représente la case de la quatrième colonne dans la troisième ligne du tableau T, c'est-à-dire celle correspondant à T(2, 3).

Bien entendu, comme pour les listes, il est nécessaire que la valeur de l'expression 1 ne dépasse pas celle du premier nombre donné, entre parenthèses, dans le dimensionnement du tableau (ou 10 si ce dimensionnement a été omis) et que la valeur de l'expression 2 ne dépasse pas celle du second nombre dans ce dimensionnement (ou 10 lorsque ce dernier a été omis).

Une variable indicée de tableau représente une seule valeur et peut être utilisée aux mêmes endroits qu'une variable simple ou une variable indicée de liste. Dans tout ce qui précède, il faut entendre par variable, soit une variable simple, soit une variable indicée de liste ou de tableau.

Exemples :

1)

```
 10  DIM T(1, 2)
 20  INPUT T(0, 0)
 30  INPUT T(1, 0)
 40  LET T(0, 1) = T(0, 0) ↑ 2
 50  LET T(1, 1) = T(1, 0) ↑ 2
 60  LET T(0, 2) = T(0, 0) + T(0, 1)
 70  LET T(1, 2) = T(1, 0) + T(1, 1)
 80  PRINT T(0, 0), T(0, 1), T(0, 2)
 90  PRINT T(1, 0), T(1, 1), T(1, 2)
100  END
```

A la ligne 10, on dimensionne un tableau de deux lignes numérotées 0 et 1 et de 3 colonnes numérotées 0, 1 et 2.

Aux lignes 20 et 30, on lit à partir du terminal les valeurs qui seront placées respectivement dans la case de la ligne 0 et de la colonne 0, et dans la case de la ligne 1 et de la colonne 0 ; ce sont donc deux valeurs placées dans la colonne 0 du tableau T.

Aux lignes 40 et 50, on affecte, à chaque case de la colonne 1 du tableau le carré de la valeur se trouvant à la même ligne dans la colonne précédente. Puis aux lignes 60 et 70, on place dans chaque case de la colonne 2, la somme des valeurs se trouvant dans la même ligne et dans les deux colonnes précédentes.

Si *v* est la valeur lue à la ligne 10 et *w* celle lue à la ligne 20,

le tableau T a la forme suivante après l'exécution des lignes 20 à 70.

	0	1	2
0	v	v^2	$v + v^2$
1	w	w^2	$w + w^2$

Enfin, aux lignes 80 et 90, on imprime le tableau avec la même disposition que ci-dessus (sans indication bien sûr des numéros de lignes et de colonnes).

2) Les boucles POUR imbriquées sont particulièrement utiles dans la manipulation des tableaux.

Ecrivons par exemple un programme qui lit à partir d'une zone de données les éléments d'un tableau T de 4 lignes et de 6 colonnes. Nous remplissons le tableau ligne par ligne : le numéro de la ligne qui est remplie varie de 0 à 3. Nous noterons I cette variable.

Le remplissage de la ligne I doit se faire élément par élément, c'est-à-dire qu'il faut d'abord lire une valeur dans la zone de données et l'affecter au premier élément de la ligne I, c'est-à-dire à T(I, 0), puis lire la valeur suivante et l'affecter au second élément de la ligne I, donc à T(I, 1), etc. Plus généralement, il faut lire six valeurs consécutives dans la zone de données, et affecter la (J + 1 ième) valeur lue à l'élément T(I, J).

La lecture de la I-ième ligne peut donc être effectuée par la boucle POUR suivante :

```
20  FØR J = 0 TØ 5
30  READ T(I, J)
40  NEXT J
```

Pour lire les 4 lignes, il faut maintenant « englober » cette boucle POUR dans une autre, qui fait varier I de 0 à 3. La lecture du tableau total peut donc être décrite de la manière suivante :

```
10  FØR I = 0 TØ 3
20  FØR J = 0 TØ 5
30  READ T(I, J)
40  NEXT J
50  NEXT I
```

Remarquons que si on intervertit les lignes 10 et 20 et les lignes 40 et 50, les éléments du tableau T seront introduits colonne par colonne et non plus ligne par ligne :

```
10  FØR J = 0 TØ 5
20  FØR I = 0 TØ 3
30  READ T(I, J)
40  NEXT I
50  NEXT J
```

3) Donnons un dernier exemple d'utilisation combinée d'une liste et d'un tableau :

Cinq magasins distribuent trois produits. Le prix de chaque produit est donné dans une liste P. La quantité vendue par produit dans chaque magasin est donnée dans un tableau Q : Q(I, J) représente la quantité vendue du produit J par le magasin I.

On suppose que la liste P et le tableau Q sont connus et on demande d'imprimer le montant total des ventes de chaque magasin. Notons V ce montant. Pour le magasin I, ce sera :

$$V = P(1) \times Q(I, 1) + P(2) \times Q(I, 2) + P(3) \times Q(I, 3)$$

En fait, il suffit de calculer $P(J) \times Q(I, J)$ pour $J = 1, 2, 3$ et d'additionner les résultats obtenus.

Une méthode simple consiste à cumuler ces valeurs dans V, c'est-à-dire d'écrire $V = V + P(J) \times Q(I, J)$ et d'effectuer cette affectation pour J valant 1 puis 2 puis 3.

On prend zéro comme valeur initiale de V, de telle sorte que V vaut $P(1) \times Q(I, 1)$ après la première affectation, puis $P(1) \times Q(I, 1) + P(2) \times Q(I, 2)$ après la seconde, etc.

```
10  FØR I = 1 TØ 5
20  LET V = 0
30  FØR J = 1 TØ 3
40  LET V = V + P(J) * Q(I, J)
50  NEXT J
60  PRINT " TØTAL VENTES MAGASIN " I; " = "; V
70  NEXT I
80  END
```

IX.5. Instructions matricielles

Il est aussi possible d'effectuer certaines opérations, globalement sur toutes les valeurs d'une liste ou d'un tableau (on parlera plus brièvement de *matrice*), grâce aux instructions matricielles.

Reprenons l'exemple 2 du § IX.2. Le programme correspondant qui calcule et imprime la somme des quantités vendues de deux articles mois par mois peut être écrit plus simplement :

```
10  DIM V(12), X(12), S(12)
20  MAT S = V + X
30  MAT PRINT S
40  END
```

La ligne 20 permet d'ajouter chaque élément de la liste V à l'élément de même numéro de la liste X et de placer le résultat dans la case correspondante de la liste S.

Elle est en fait équivalente à la boucle POUR suivante :

```
20  FØR I = 0 TØ 12
21  LET S(I) = V(I) + X(I)
22  NEXT I
```

De même, la ligne 30 permet d'imprimer globalement la liste S sur le terminal. Remarquons que cette façon de faire correspond non seulement à une écriture plus condensée mais aussi à une rapidité d'exécution plus grande que la boucle POUR équivalente.

Les instructions matricielles qui peuvent être utilisées en BASIC sont les suivantes (nous supposons dans ce qui suit que A, B et C sont trois matrices quelconques, c'est-à-dire trois variables de liste ou de tableau).

1) *Addition et soustraction* :

Elles s'écrivent :

```
   MAT C = A + B
et MAT C = A — B
```

Nous avons expliqué précédemment l'effet de la première. La seconde a un effet tout à fait analogue.

Remarquons que les variables A, B et C doivent représenter des matrices « cohérentes » ; elles doivent toutes être :

— soit des variables de liste ayant reçu le même dimensionnement,
— soit des variables de tableau ayant reçu le même dimensionnement, c'est-à-dire ayant le même nombre de lignes et le même nombre de colonnes. Il est par exemple, interdit d'écrire :

```
10  DIM A(15), B(3, 12), C(8)
20  MAT C = A + B
```

Il est de même interdit d'écrire :

```
10  DIM A(5), B(7), C(12)
20  MAT C = A + B
```

2) *Multiplication (au sens mathématique)* :

Elle s'écrit :

```
MAT C = A * B
```

La variable A doit nécessairement être une variable de tableau, mais la matrice B peut être une variable de liste ou de tableau.

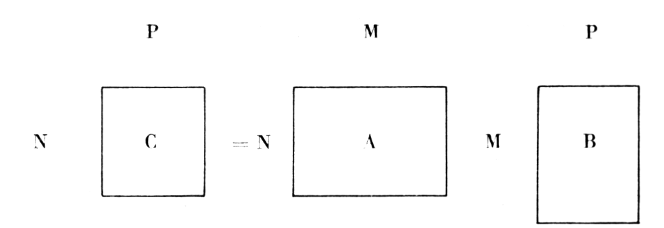
D'autre part, la variable C doit être différente de A et de B, mais on peut par exemple écrire :

```
MAT C = A * A
```

Si le tableau A contient M colonnes, la matrice B doit contenir M lignes. Remarquons qu'une liste B de M éléments est considérée comme un tableau d'une colonne et de M lignes.

D'autre part, si la matrice B contient P colonnes (éventuellement P est égal à 1 si B est une liste) et si la matrice A contient N lignes, alors la matrice C doit contenir N lignes et P colonnes.

En d'autres termes, la matrice C doit contenir autant de lignes que A et autant de colonnes que B.



L'instruction MAT C = A ⋇ B est alors équivalente aux trois boucles POUR imbriquées suivantes (1) :

```
FØR I = 0 TØ N — 1
FØR J = 0 TØ P — 1
LET C(I, J) = 0
FØR K = 0 TØ M — 1
LET C(I, J) = C(I, J) + A(I, K) ⋇ B(K, J)
NEXT K
NEXT J
NEXT I
```

Ainsi le nouvel élément C(I, J) est égal à la somme :

A(I, 0) ⋇ B(0, J) + A(I, 1) ⋇ B(1, J) + . . .
+ A(I, M — 1) ⋇ B(M — 1, J)

Remarques :

— d'après ce que nous venons de dire, C sera une liste si B en est une.
— après avoir écrit le dimensionnement suivant :

```
10 DIM C(4), A(4, 6), B(6)
```

on peut écrire :

```
20 MAT C = A ⋇ B
```

(1) Rappelons que les lignes et les colonnes sont numérotées à partir de 0, donc jusqu'à N — 1, M — 1 et P — 1 dans l'exemple pris.

mais on ne peut pas écrire

```
30  MAT C = B * A
```

Pour pouvoir multiplier une liste B par une matrice, dans cet ordre, il faut que B soit dimensionné comme un tableau à une seule ligne, c'est-à-dire par exemple au moyen de :

```
10  DIM B(0, 4)
```

Dans ce cas, la multiplication B * A fournit comme résultat un tableau à une seule ligne : c'est en fait une liste d'éléments « placés en ligne ».

Exemple : L'écriture suivante est correcte :

```
10  DIM C(0, 6), A(4, 6), B(0, 4)
20  MAT C = B * A
```

3) *Multiplication de tous les éléments par une valeur :*

Il est possible aussi de multiplier chacun des éléments d'une matrice par une même valeur, résultat d'une expression, et d'affecter ces éléments ainsi multipliés à une autre matrice. L'écriture de cette instruction est la suivante :

```
MAT C = (expression) * A
```

A et C doivent être toutes deux des listes ou des tableaux de même dimensionnement. Soit N ce dimensionnement ; l'instruction est équivalente à :

```
FØR I = 0 TØ N
LET C(I) = (expression) * A(I)
NEXT I
```

Si C et A sont des tableaux contenant tous deux N lignes et M colonnes, l'instruction MAT C = (expression) * A est équivalente à :

```
FØR I = 0 TØ N — 1
FØR J = 0 TØ M — 1
LET C(I, J) = (expression) * A(I, J)
NEXT J
NEXT I
```

Remarques : L'instruction qui permet d'affecter globalement une matrice A à une matrice C n'existe pas en BASIC. On ne peut pas écrire :

MAT C = A

Cependant, la difficulté est aisément tournée en écrivant :

MAT C = (1) * A

D'autre part, il est interdit de faire dans une même instruction matricielle plusieurs opérations successives. Par exemple, on ne peut pas écrire :

```
10   MAT C = A * B + D
```

Cette difficulté est aussi aisément tournée ; dans ce cas particulier, en écrivant :

```
10   MAT C = A * B
20   MAT C = C + D
```

4) *Transposition et inversion* (au sens mathématique) :

Ces opérations ne peuvent être effectuées que sur des tableaux « carrés », c'est-à-dire ayant même nombre de lignes et de colonnes.

Elles s'écrivent respectivement :

MAT C = TRN(A)

et

MAT C = INV(A)

Le tableau C doit être différent de A et avoir le même dimensionnement (en particulier il doit lui aussi être carré).

La première affecte à C le tableau transposé de A et la seconde la matrice inverse de A.

5) *Initialisations* :

Il est possible de donner des valeurs particulières aux éléments d'un tableau, en redimensionnant éventuellement ce tableau, c'est-à-dire en changeant son nombre de lignes et son

nombre de colonnes. Les instructions correspondantes sont de l'une des deux formes suivantes :

MAT C = initialisation

ou

MAT C = initialisation (expression 1, expression 2).

Dans le premier cas, les éléments de C reçoivent les valeurs déterminées par l'initialisation. En outre, dans le second cas, si $v1$ est la valeur de l'expression 1 et $v2$ celle de l'expression 2, seuls les éléments dans les $v1 + 1$ premières lignes (numérotées de 0 à $v1$) et les $v2 + 1$ premières colonnes (numérotées de 0 à $v2$) sont initialisés. Les autres éléments du tableau C sont alors inaccessibles dans la suite du traitement. Remarquons que les valeurs $v1$ et $v2$ doivent être au plus égales aux valeurs correspondant au dimensionnement du tableau C au moment de l'exécution de cette instruction.

Les initialisations peuvent être de trois types :

— initialisation à zéro : elle correspond au sigle ZER.

Par exemple : MAT C = ZER.

Dans ce cas, les éléments de C sont mis à zéro.

— initialisation à un : elle correspond au sigle CON.

Par exemple : MAT C = CON.

Dans ce cas, les éléments de C sont mis à un.

— initialisation à une matrice unité : elle correspond au sigle IDN.

Par exemple : MAT C = IDN.

Le tableau C doit alors être carré. Ses éléments sont mis à zéro sauf ceux de la diagonale principale (de la forme C(I, I)) qui sont mis à un.

6) *Lecture à partir d'une zone de données :*

Elle peut se faire avec ou sans redimensionnement du tableau au moyen de l'une des instructions suivantes :

MAT READ C

ou

MAT READ C (expression 1, expression 2)

Dans le second cas, le tableau C est redimensionné de la même façon que celle indiquée pour les initialisations. Le remplissage du tableau C se fait ligne par ligne.

Pour un tableau de N lignes et de M colonnes, cette instruction est donc équivalente à :

```
FØR I = 0 TØ N — 1
FØR J = 0 TØ M — 1
READ C(I, J)
NEXT J
NEXT I
```

Il est aussi possible de mettre plusieurs variables de tableau, redimensionnées ou non, dans une seule instruction MAT READ.

Exemple :

```
10  MAT READ A, B(10, N + 1), C, D
```

Les tableaux A, B, C et D sont lus, comme il a été indiqué ci-dessus, les uns après les autres.

7) *Impression* :

L'instruction correspondante est de la forme suivante : MAT PRINT liste de variables de tableau.

Les variables de tableau de la liste peuvent être séparées par des virgules ou des points-virgules.

Chaque tableau est imprimé ligne par ligne et l'effet des séparateurs entre les différentes variables est le même que pour l'instruction PRINT (voir chap. IV).

EXERCICES

— Quelle différence faites-vous entre variable simple et variable simple indicée ou liste ?

— L'écriture
A(B$(I))
est-elle permise ? Pourrait-elle donner lieu à un résultat, à un calcul ?

— L'écriture :

B$(A(I))

est-elle permise ? Est-elle équivalente à celle de l'exercice précédent ?

— Chercher la somme de dix variables : $V_1, \ldots, V_{10}$.
Applications : $V_i = 2\,000$. $i = 1, \ldots, 10$.

— Comment sont rangées en mémoire de l'ordinateur les variables de tableau ?

— Quel est l'effet du dimensionnement ?
Est-il toujours indispensable en langage BASIC ?

— On vous donne deux listes de nombres S1 et S2.
Chercher les éléments de même rang égaux dans les deux listes et les ranger dans une troisième liste S. Organigramme et programme.

— On a regroupé dans une variable de liste cent observations statistiques d'un phénomène.
— Rechercher la plus faible observation, la plus grande.
— Déterminer l'étendue de la population (plus grande moins plus faible).
Les observations se répartissent en dix classes dont l'amplitude sera un paramètre. On vous demande d'écrire le programme déterminant le nombre d'éléments par classe.

— Considérer une liste dont certains articles sont nuls. On vous demande de dresser l'organigramme et d'écrire le programme permettant de les éliminer et de mettre la liste à jour.

Chapitre X

LES SOUS-PROGRAMMES

Nous avons vu au chapitre III que, lorsqu'une expression est écrite en plusieurs endroits d'un programme, il est plus simple et plus court de la représenter par une fonction et de remplacer aux dits endroits du programme cette expression par un appel de fonction.

Dans certains problèmes, une situation analogue se présente, non plus pour une expression mais pour plusieurs instructions d'un programme. Par exemple, la mise à jour d'un stock en tenant successivement compte des entrées puis des sorties, nécessite la recherche d'un article en stock. Cette recherche ne se fera pas au même moment selon que l'article fait l'objet d'une entrée ou d'une sortie. Il lui correspondra donc deux suites d'instructions identiques en deux endroits différents du programme : l'une pour l'entrée, l'autre pour la sortie de l'article.

Dans un tel cas, il est plus agréable de n'écrire qu'une seule fois la séquence d'instructions qui apparaît en plusieurs endroits, et de la remplacer, là où elle devrait figurer, par une seule instruction indiquant à l'ordinateur qu'il doit effectuer toute la séquence d'instructions omises.

Une telle séquence d'instructions, écrite une seule fois, mais pouvant être exécutée en plusieurs endroits différents d'un programme, est appelée un *sous-programme*. Par opposition, nous dirons que le programme qui utilise des sous-programmes est le programme principal. L'utilisation des sous-programmes permet d'écrire des programmes plus courts, qui prendront donc

moins de place en mémoire de l'ordinateur. En outre, elle facilite la modification des programmes. En effet, s'il faut modifier une partie d'un traitement, représentée par un sous-programme, il suffira uniquement de modifier ce sous-programme. Les instructions qui permettent d'exécuter le sous-programme en divers endroits du programme principal, seront inchangées. Par contre, si un sous-programme n'avait pas été utilisé, le programme principal contiendrait plusieurs fois la même séquence d'instructions et il faudrait modifier chacune de ces séquences.

Le but du présent chapitre est de présenter deux types de sous-programmes utilisés en BASIC.

X.1. Sous-programmes GØSUB

Une première possibilité d'emploi des sous-programmes consiste à écrire les séquences d'instructions correspondantes après le programme principal, c'est-à-dire après la ligne END du programme principal.

Un sous-programme est constitué d'une succession de lignes contenant n'importe quelles instructions permises en BASIC. Ces lignes doivent avoir un numéro supérieur à celui de la ligne END du programme principal. L'ordre de ces lignes a exactement la même signification que pour un programme normal.

La dernière ligne d'un sous-programme doit nécessairement comporter le mot RETURN.

Exemple :

```
 10
  .
  .   « Programme principal »
  .
100   END
110   INPUT X
120   LET X = 2 * Y / X
130   PRINT " X = " X
140   RETURN
```

Les lignes 110 à 140 décrivent un sous-programme qui peut être utilisé dans le programme principal (lignes 10 à 100). Lorsqu'il sera exécuté (voir ci-dessous), ce sous-programme entrera une valeur pour la variable X, à partir du terminal, calculera une nouvelle valeur pour X (à partir de la valeur entrée et de la valeur que la variable Y aura à ce moment), enfin imprimera cette nouvelle valeur.

Un programme principal peut utiliser plusieurs sous-programmes. Pour cela, il suffit de les écrire les uns à la suite des autres, après le programme principal : la première ligne du premier sous-programme suit immédiatement la ligne END du programme principal, la première ligne du second sous-programme suit immédiatement la ligne RETURN du premier sous-programme et ainsi de suite.

L'exécution d'un sous-programme ne peut pas être interrompue par une rupture de séquence ou une instruction conditionnelle qui renvoie dans le programme principal. Par exemple, le programme suivant est incorrect.

```
 10
  .
  .    « Programme principal »
  .
100  END
110  INPUT X
120  IF X > 0 THEN 140
130  GØTØ 60
140  LET X = 2 * Y / X
150  PRINT " X = " X
160  RETURN
```

En effet, lorsque l'ordinateur exécutera le sous-programme, il devra retourner à la ligne 60, si la valeur entrée pour X est négative ou nulle. Cela est interdit : le déroulement du sous-programme doit nécessairement se terminer par l'instruction RETURN.

Par contre, il est possible d'exécuter des ruptures de séquence qui conduisent à l'intérieur d'un même sous-programme, ou même d'un autre sous-programme.

Ainsi, les deux « programmes » suivants sont corrects :

```
 10
 .
 .    « Programme principal »
 .
100   END
110   INPUT X
120   IF X > 0 THEN 150
130   LET X = 2 * Y / X
140   GØTØ 160
150   LET X = Y / X + Z
160   PRINT " X = " X
170   RETURN
```

```
 10
 .
 .    « Programme principal »
 .
100   END
110   INPUT X
120   IF X > 0 THEN 170
130   LET X = 2 * Y / X
140   PRINT " X = " X
150   RETURN
160   LET X = X + 1
170   INPUT Z
180   LET X = Y / X + Z
190   RETURN
```

Dans le second exemple, les lignes 110 à 150 représentent un premier sous-programme, et les lignes 160 à 190 un second.

Si, lors de l'exécution du premier sous-programme, la valeur lue pour X est positive, l'ordinateur exécutera les instructions des lignes 170 et 180 avant de « revenir » au programme principal (voir ci-dessous).

Remarque : Il n'est pas toujours nécessaire d'écrire les sous-programmes après la ligne END du programme principal. En effet, un sous-programme peut être écrit après n'importe quelle

instruction GØTØ du programme principal. Cependant, nous déconseillons cette pratique, le programme devenant alors difficile à lire.

Pour utiliser un sous-programme dans le programme principal, il suffit d'utiliser l'instruction d'appel de sous-programme dont le schéma est le suivant :

GØSUB numéro de ligne.

Lorsque l'ordinateur rencontre une telle instruction, il exécute l'instruction dont le numéro de ligne est indiqué après GØSUB, puis les instructions des lignes qui la suivent (c'est-à-dire des différentes lignes du sous-programme) jusqu'à ce qu'il rencontre l'instruction RETURN. Il « revient » alors au programme principal et exécute l'instruction située à la ligne du programme principal qui suit celle où figurait l'instruction GØSUB.

Tout se passe comme si l'ordinateur remplaçait l'instruction GØSUB par le sous-programme correspondant et changeait les instructions RETURN de ce sous-programme par une instruction GØTØ au numéro de ligne qui suit celui de l'instruction GØSUB.

Exemples :

```
 10  INPUT Y
 20  GØSUB 110
 30  IF X = 10 THEN 60
  .
  .
  .
100  END
110  INPUT X
120  LET X = 2 * Y / X
130  PRINT " X = " X
140  RETURN
```

Quand l'ordinateur rencontre la ligne 20, il exécute les instructions des lignes 110, 120 et 130, puis revient au programme principal et exécute l'instruction de la ligne 30. Remarquons qu'à ce moment, la valeur de X est celle calculée

à la ligne 120 du sous-programme ; d'autre part, à la ligne 120, la valeur de Y est celle qui a été lue à la ligne 10. Ainsi, les variables sont « communes » au programme principal et au sous-programme, c'est-à-dire qu'elles ont la même signification et que leurs valeurs dépendent des instructions qui ont été exécutées précédemment, que ces instructions soient dans le programme principal ou dans un (ou des) sous-programme(s).

```
 10  INPUT Y
 20  GØSUB 110
 30  IF X = 10 THEN 60
  .
  .
  .
100  END
110  INPUT X
120  IF X > 0 THEN 170
130  LET X = 2 * Y / X
140  PRINT " X = " X
150  RETURN
160  LET X = X + 1
170  INPUT Z
180  LET X = Y / X + Z
190  RETURN
```

Dans cet exemple, l'ordinateur exécute l'instruction de la ligne 110 après celle de la ligne 20. Puis, si la valeur lue pour X est positive, il exécute les instructions des lignes 170 et 180 avant de revenir au programme principal, c'est-à-dire à la ligne 30 ; sinon il exécute les instructions des lignes 130 et 140, et revient au programme principal.

Il est bien sûr possible de faire plusieurs appels d'un sous-programme, en des endroits différents du programme principal. Remarquons que dans ce cas, les valeurs des variables que le sous-programme utilise pourront avoir changé.

Exemple :

```
 10  INPUT Y
 20  GØSUB 110
```

```
 30  IF X = 10 THEN 60
 40  LET Y = Y + 1
 50  GØSUB 110
  .
  .
  .
100  END
110  INPUT X
120  LET X = 2 * Y / X
130  PRINT " X = " X
140  RETURN
```

Lors du second appel de la ligne 50, la valeur de Y n'est plus égale à ce qu'elle était lors de l'appel de la ligne 20.

Un sous-programme peut aussi appeler un autre sous-programme au moyen d'une instruction GØSUB. Le fonctionnement est le même que celui décrit ci-dessus.

Cependant, il est interdit qu'un sous-programme en appelle un autre après avoir été appelé, lui-même, par cet autre (en particulier un sous-programme ne peut pas s'appeler lui-même).

Exemple :

Le « programme » suivant est incorrect :

```
 10
  .
  .   « Programme principal »
  .
100  END
110  INPUT X
120  IF X > 0 THEN 140
130  GØSUB 160
140  LET X = 2 * Y / X
150  RETURN
160  INPUT Z
170  LET X = Y / X + Z
180  GØSUB 110
190  RETURN
```

Remarques :

1) L'appel d'un sous-programme peut renvoyer à une ligne différente de la première ligne du sous-programme.

Exemple :

```
 10  INPUT X, Y
 20  GØSUB 110
 30  LET Y = X + Y
 40  GØSUB 120
  .
  .
  .
100  END
110  LET X = X + 1
120  IF X * Y > 0 THEN 140
130  LET X = 2 * X + Y
140  PRINT (X — Y) / 2
150  RETURN
```

Lors du premier appel du sous-programme, l'ordinateur exécute l'instruction de la ligne 110, alors qu'il ne le fait pas lors du second appel.

2) Si, dans l'exécution d'un sous-programme, l'ordinateur rencontre une instruction END ou STØP, il s'arrête, comme si cette instruction figurait dans le programme principal.

Donnons un exemple pratique d'utilisation d'un sous-programme.

Etant donné trois listes X, Y et Z de 20 éléments, le problème consiste à :

— faire le produit élément par élément des listes X et Y, Y et Z, Z et X et mettre ce produit dans la liste S ;
— chercher à chaque fois le maximum des nombres de la liste S et l'imprimer s'il est supérieur à 10.

Sans utiliser de sous-programmes, ce problème peut être résolu par le programme suivant :

```
 10 DIM X(20), Y(20), Z(20), S(20)
 15 MAT READ X, Y, Z
 20 FØR I = 1 TØ 20
 30 LET S(I) = X(I) * Y(I)
 40 NEXT I
 50 LET M = S(1)
 60 FØR I = 2 TØ 20
 70 IF S(I) < = M THEN 90
 80 LET M = S(I)
 90 NEXT I
100 IF M < = 10 THEN 120
110 PRINT M
120 FØR I = 1 TØ 20
130 LET S(I) = Y(I) * Z(I)
140 NEXT I
150 LET M = S(1)
160 FØR I = 2 TØ 20
170 IF S(I) < = M THEN 190
180 LET M = S(I)
190 NEXT I
200 IF M < = 10 THEN 220
210 PRINT M
220 FØR I = 1 TØ 20
230 LET S(I) = Z(I) * X(I)
240 NEXT I
250 LET M = S(1)
260 FØR I = 2 TØ 20
270 IF S(I) < = M THEN 290
280 LET M = S(I)
290 NEXT I
300 IF M < = 10 THEN 320
310 PRINT M
320 END
```

Les séquences de lignes 20 à 40, 120 à 140 et 220 à 240 calculent les produits des listes X, Y, Z.

Les séquences de lignes 50 à 110, 150 à 210 et 250 à 310 sont analogues : elles calculent le maximum de la liste S, qui contient suivant les cas les produits des listes X et Y ou Y et Z ou Z et X, et impriment ce maximum quand il est plus grand que 10.

Ce traitement identique dans les trois cas peut faire l'objet d'un sous-programme. Le programme suivant, plus court, réalise le même travail que le précédent.

```
 10  DIM X(20), Y(20), Z(20), S(20)
 20  FØR I = 1 TØ 20
 25  INPUT X(I), Y(I), Z(I)
 30  LET S(I) = X(I) * Y(I)
 40  NEXT I
 50  GØSUB 150
 60  FØR I = 1 TØ 20
 70  LET S(I) = Y(I) * Z(I)
 80  NEXT I
 90  GØSUB 150
100  FØR I = 1 TØ 20
110  LET S(I) = Z(I) * X(I)
120  NEXT I
130  GØSUB 150
140  END
150  LET M = S(1)
160  FØR I = 2 TØ 20
170  IF S(I) < = M THEN 190
180  LET M = S(I)
190  NEXT I
200  IF M < = 10 THEN 220
210  PRINT M
220  RETURN
```

X.2. Sous-programmes CALL

Les sous-programmes GØSUB permettent de réduire et de simplifier l'écriture d'un programme mais présentent l'inconvénient d'être compilés, c'est-à-dire traduits en langage-machine

(voir chap. I), en même temps que le programme principal. Cela est un inconvénient pour deux raisons :

— il est nécessaire que les traductions en langage-machine du programme principal et des sous-programmes GØSUB qui lui sont attachés, soient faites en une seule fois ; or, le nombre d'instructions BASIC qui peuvent être compilées en une seule fois est limité ; cette limite est plus stricte que celle concernant le nombre d'instructions-machine résultant d'une compilation et pouvant être gardées en même temps dans la mémoire de l'ordinateur ;

— dans de nombreux problèmes pratiques, plusieurs programmes différents utilisent des sous-programmes identiques. Si nous ne pouvions employer que des sous-programmes GØSUB, il faudrait écrire autant de sous-programmes qu'il y a de programmes principaux qui les utilisent, puisqu'un sous-programme GØSUB est attaché à un programme principal et un seul.

Pour pallier ces inconvénients, il est possible d'utiliser en BASIC un autre type de sous-programmes : les sous-programmes CALL.

Un sous-programme CALL est constitué d'une succession quelconque de lignes BASIC, *qui peuvent être écrites indépendamment de tout programme principal*. Il constitue un tout et ne diffère d'un programme ordinaire que par le fait que sa dernière ligne contient une instruction RETURN et non une instruction END.

Exemple :

```
10  LET M = S(1)
20  FØR I = 2 TØ 20
30  IF S(I) < = M THEN 50
40  LET M = S(I)
50  NEXT I
60  IF M < = 10 THEN 80
70  PRINT M
80  RETURN
```

Ce sous-programme calcule le maximum d'une liste S de 20 éléments (cette liste peut être quelconque) et l'imprime s'il est plus grand que 10.

Pour pouvoir utiliser un sous-programme CALL, il faut lui donner un nom et l'avoir stocké préalablement sous ce nom comme un programme ordinaire (voir chap. I).

Lorsque ces opérations ont été faites, n'importe quel programme peut utiliser ce sous-programme CALL, au moyen d'une instruction dont le schéma est le suivant :

CALL nom du sous-programme.

Exemple : En supposant que le sous-programme de l'exemple précédent ait été appelé MAXIMUM, l'exemple de la fin du § X.2 s'écrira simplement :

```
 10  DIM X(20), Y(20), Z(20), S(20)
 20  FØR I = 1 TØ 20
 25  INPUT X(I), Y(I), Z(I)
 30  LET S(I) = X(I) * Y(I)
 40  NEXT I
 50  CALL MAXIMUM
 60  FØR I = 1 TØ 20
 70  LET S(I) = Y(I) * Z(I)
 80  NEXT I
 90  CALL MAXIMUM
100  FØR I = 1 TØ 20
110  LET S(I) = Z(I) * X(I)
120  NEXT I
130  CALL MAXIMUM
140  END
```

Lors de l'exécution de l'instruction à la ligne 50, le sous-programme MAXIMUM sera introduit en mémoire centrale de l'ordinateur et compilé. Remarquons qu'à ce moment, le programme principal qui appelle MAXIMUM a déjà été entièrement compilé et la limitation dont nous avons parlé plus haut, ne porte donc plus que sur les instructions à compiler pour le sous-programme MAXIMUM. Ainsi, il est possible que l'ordinateur ne puisse pas compiler le programme de la fin du § X.2,

alors qu'il pourra compiler *séparément* le programme ci-dessus et le sous-programme MAXIMUM.

Lorsque le sous-programme MAXIMUM a été compilé, l'ordinateur l'exécute, puis quand il rencontre l'instruction RETURN, il revient au programme principal comme pour un sous-programme GØSUB.

Remarquons que lors du second et du troisième appel de MAXIMUM aux lignes 90 et 130, le sous-programme MAXIMUM ne sera pas recompilé : l'ordinateur a gardé en mémoire la traduction en langage-machine résultant de l'appel de la ligne 50.

Remarques :

1) Le sous-programme MAXIMUM peut être appelé par d'autres programmes que celui que nous avons écrit ci-dessus. Dans tous les cas, le mécanisme est le même.

2) Un programme peut bien sûr appeler plusieurs sous-programmes CALL différents.

3) Un sous-programme CALL peut appeler des sous-programmes GØSUB, qui lui sont attachés (comme pour un programme ordinaire) ; il peut même appeler d'autres sous-programmes CALL, avec les mêmes limitations que celles indiquées pour les sous-programmes GØSUB.

4) Il n'est pas possible de « sortir » d'un sous-programme CALL par une rupture de séquence ou une instruction conditionnelle qui renvoie à une ligne ne figurant pas dans ce sous-programme.

5) Si, lors de l'exécution d'un sous-programme CALL, l'ordinateur rencontre une instruction END ou STØP, il s'arrête.

6) Toutes les variables et les fonctions définies dans un programme principal et dans des sous-programmes CALL sont communes. En particulier, lors d'un appel d'un sous-programme CALL, les variables qui figurent dans le sous-programme ont comme valeurs celles qui résultent des calculs dans le programme qui appelle ce sous-programme.

7) Les zones de données (voir chap. V) du programme principal et des sous-programmes CALL sont regroupées : lors du premier appel d'un sous-programme CALL, la zone de données qu'il contient éventuellement vient se placer après celle qui avait déjà été constituée pour le programme principal, en tenant compte, le cas échéant, d'appels antérieurs d'autres sous-programmes CALL.

Exemple : Si le sous-programme suivant s'appelle SP :

```
10  READ X
20  DATA 4
30  RETURN
```

l'exécution du programme principal

```
10  READ A, B
20  CALL SP
30  READ C
40  PRINT A, B, X, C
50  DATA 1, 2, 3
60  END
```

imprimera successivement les valeurs 1, 2, 4 et 3.

EXERCICES

— Quelle est la dernière instruction d'un sous-programme de type GØSUB ? De type CALL ?

— Quelles différences existe-t-il entre un sous-programme CALL et un sous-programme GØSUB ? Pensez-vous que ce sont uniquement des avantages ?

— Pensez-vous que les fonctions du type
DEF FNX(X)
peuvent être considérées comme des sous-programmes ?
Quelles sont les restrictions ?

— Par une rupture de séquence conditionnelle ou non, pouvez-vous quitter un sous-programme pour retourner au programme principal dit encore programme appelant ? Est-il possible d'exécuter, à partir d'un sous-programme, une rupture de séquence vers un autre sous-programme ?

Chapitre XI

LES FICHIERS

La plupart des problèmes de gestion et certains problèmes scientifiques nécessitent le traitement d'un grand nombre d'informations. La gestion d'un stock de mille articles, chacun d'eux étant caractérisé par cinquante caractères, nécessitera le traitement de cinquante mille caractères.

Il est hors de question et très souvent impossible par manque de place, de garder toutes ces informations globalement en mémoire centrale de l'ordinateur.

Or, chaque traitement qui met à jour de telles informations prend comme données de départ, celles issues du traitement précédent. Il est donc indispensable de garder ces informations entre deux traitements pour les utiliser ultérieurement.

Une première solution envisageable consiste en l'utilisation d'un support tel qu'un ruban ou des cartes perforés. Elle présente de graves inconvénients : le premier est que, avant chaque nouveau traitement, il faudra lire toutes les cartes perforées (ou le ruban perforé) contenant les informations du traitement précédent. En outre, à la fin du traitement, il faudra perforer de nouvelles cartes (ou un nouveau ruban) pour garder les informations qui viennent d'être modifiées. Or, ces temps de lecture et de perforation sont le plus souvent très longs.

Qui plus est, il est impossible de lire en une seule fois toutes les cartes perforées (ou le ruban perforé) si les informations correspondantes ne « tiennent » pas en mémoire centrale de l'ordinateur. Il faudra alors en lire une partie, traiter les informations correspondantes, perforer les résultats obtenus, puis recommencer avec une autre partie des anciennes informations et ainsi de suite. Il s'ensuit le second inconvénient à la solution envisagée : elle est absolument impraticable si le traitement

d'une partie quelconque des anciennes informations exige de connaître ou d'avoir accès rapidement à n'importe laquelle de ces informations.

L'autre solution, qui résout ces problèmes, est de garder toutes les informations sur des mémoires périphériques telles que des disques ou des bandes magnétiques. Cette solution présente l'avantage de la rapidité de lecture ou d'écriture d'une information. En outre, elle permet, comme nous le verrons plus loin, d'accéder à tout moment à une information quelconque.

Les informations sont stockées sur un disque ou une bande magnétique dans ce que nous appellerons un *fichier*, c'est-à-dire une succession d'éléments placés les uns à la suite des autres dans un certain ordre. Chaque élément est constitué d'une série d'informations qui le caractérise. Par exemple, chaque élément d'un fichier stock regroupe toutes les informations concernant un article en stock, telles que sa désignation, son prix de vente, son prix de revient, la quantité en stock de cet élément, etc.

Le but du présent chapitre est de décrire les différentes utilisations possibles d'un fichier, puis de donner deux types très généraux de fichiers et la façon de les manipuler en BASIC.

XI.1. Utilisations des fichiers

Les traitements sur les fichiers sont essentiellement de quatre types :

— recherche d'une ou de plusieurs informations relatives à un ou plusieurs éléments.

Exemple : La valorisation d'un stock au prix de revient nécessite la recherche pour chaque élément, de la quantité en stock et du prix de revient ;

— modification d'une ou de plusieurs informations relatives à un ou plusieurs éléments.

Exemple : La mise à jour d'un stock nécessite la modification de la quantité en stock, des quantités entrées et sorties..., de certains éléments ;

— classement des éléments dans un certain ordre.

Exemple : Classement des articles en stock par prix de vente décroissants ;

— ajout ou retrait d'un ou de plusieurs éléments.

Un retrait nécessite le plus souvent de « retasser » les éléments restants, pour qu'ils soient toujours consécutifs dans le fichier modifié. Un ajout peut obliger à placer les nouveaux éléments à leur « bonne » place, si le fichier est classé dans un certain ordre.

Ces quatre traitements ne peuvent se faire que si le problème suivant est résolu :

Connaissant des informations qui déterminent parfaitement un élément du fichier, par exemple sa désignation, ou un code, retrouver :

— la place de l'élément dans le fichier ;
— l'information (ou les informations) de cet élément qu'on désire connaître ou modifier.

Cette recherche est un élément fondamental car son temps conditionne pratiquement celui d'un traitement qui utilise des fichiers. La façon dont elle est effectuée, définit deux grandes classes de fichiers :

— les fichiers séquentiels ;
— les fichiers à accès direct.

XI.2. Les fichiers séquentiels

D'une manière générale, l'accès à une information quelconque d'un fichier séquentiel nécessite d'accéder d'*abord* à chacune des informations qui la précèdent dans ce fichier.

En particulier, il n'est pas possible de lire ou d'écrire directement une information à un endroit quelconque d'un fichier séquentiel. La lecture (ou l'écriture) doit commencer au premier élément du fichier et continuer élément par élément. De plus, les informations d'un élément doivent être lues ou écrites les unes après les autres dans l'ordre où elles se trouvent dans le fichier.

Par exemple, il est impossible de lire directement la qua-

trième information du vingtième élément, sans avoir au préalable lu toutes les informations qui la précèdent.

En outre, quand une information vient d'être lue, on ne peut lire immédiatement après, que l'information qui la suit, ou éventuellement la première information du fichier en recommençant la lecture au début. Il en est de même pour l'écriture d'informations sur le fichier.

Par ailleurs, il est impossible d'alterner des lectures et des écritures sans recommencer au début du fichier. Ainsi, lorsqu'une information vient d'être lue, on ne peut pas en écrire une autre immédiatement après, sans revenir au début du fichier, c'est-à-dire qu'il faudrait écrire dans la première position du fichier. Inversement, lorsqu'une information vient d'être écrite, on ne peut en lire une autre immédiatement après sans revenir au début du fichier, c'est-à-dire qu'il faut alors lire la première information.

Il peut sembler que le temps de recherche d'un élément dans un fichier séquentiel sera long, surtout si cet élément se trouve à la fin du fichier. En fait, l'expérience prouve que si la recherche porte sur plus de 20 % des informations du fichier, l'utilisation d'un fichier séquentiel est préférable, compte tenu de la grande vitesse de transmission dans ce cas des informations depuis une mémoire périphérique vers la mémoire centrale de l'ordinateur.

Il existe en outre différentes techniques (voir bibliographie [3]) qui permettent d'accélérer le temps de recherche d'un élément.

XI.3. Utilisation des fichiers séquentiels en BASIC

Pour utiliser un fichier dans un programme, il faut auparavant le préparer, c'est-à-dire :

— le créer ;
— lui donner un nom ;
— le sauvegarder ;
— y mettre éventuellement des informations, chacune d'elles pouvant être un nombre ou une chaîne.

Ces opérations s'effectuent au moyen du langage de dialogue (voir chap. I). Nous ne les détaillerons pas car elles dépendent beaucoup des systèmes de temps partagé.

Lorsqu'elles ont été faites, le fichier peut être utilisé dans un programme BASIC quelconque.

Il existe en BASIC deux types d'utilisation d'un fichier séquentiel :

— lecture d'une information dans le fichier ;
— écriture d'une information à une place bien définie dans le fichier.

L'endroit où l'information est lue, ou la place où elle est écrite, sont repérés par un pointeur dont le rôle est analogue à celui de la zone de données (voir chap. V).

Pour utiliser un fichier, il faut d'abord l' « ouvrir », c'est-à-dire le rendre disponible à la lecture ou à l'écriture. Cette « ouverture » est réalisée par une instruction dont la forme dépend du système de temps partagé. En outre, les instructions qui permettent de lire ou d'écrire une information dans un fichier séquentiel varient d'un système à l'autre : nous décrivons ci-dessous les deux formes les plus courantes d'utilisation d'un fichier séquentiel en BASIC.

Première forme : L'ouverture de fichiers se fait au moyen d'une instruction dont le schéma est le suivant :

FILES liste de noms de fichiers.

Dans la liste, les noms des différents fichiers doivent être séparés par des points-virgule.

Exemple :

```
10 FILES STØCK ; CLIENTELE ; MAGASINS
```

Un même programme peut contenir plusieurs « instructions FILES » mais elles doivent toute précéder n'importe quelle autre instruction du programme. En particulier, il est impossible de mettre une instruction FILES dans un sous-programme CALL (voir chap. X).

Chacun des fichiers figurant dans une instruction FILES doit avoir été préalablement préparé (voir plus haut).

L'effet de cette instruction est triple :

— elle définit l'ordre des fichiers qui seront utilisés dans le programme : le premier fichier dans la liste de l'instruction FILES, dont le numéro de ligne est le plus petit, reçoit le numéro d'ordre 1 ; le second fichier de cette liste reçoit le numéro d'ordre 2 et ainsi de suite jusqu'à épuisement des fichiers de cette liste, puis en recommençant pour l'instruction FILES dont le numéro de ligne est immédiatement supérieur, etc.

Exemple :

```
10  FILES STØCK ; CLIENTELE
20  FILES MAGASINS ; AUX 1 ; AUX 2
30  FILES RESULT
```

Les fichiers STØCK, CLIENTELE, MAGASINS, AUX 1, AUX 2 et RESULT ont respectivement les numéros d'ordre 1, 2, 3, 4, 5 et 6.

Nous verrons plus loin l'importance de cette numérotation ;

— elle met chacun des fichiers de la liste dans l'*état de lecture*, c'est-à-dire qu'après cette instruction, on peut lire mais on ne peut pas écrire d'information dans ces fichiers, sans avoir au préalable fait passer leur état de lecture en écriture (voir ci-dessous). Cette restriction est importante pour éviter d'écrire intempestivement des informations, qui viendraient « détruire » celles qui figuraient précédemment dans un fichier ;

— elle positionne les pointeurs des différents fichiers de la liste sur la première information de ces fichiers. Ainsi la première lecture dans l'un quelconque de ces fichiers fournira la première information de ce fichier (voir ci-dessous).

Pour écrire une information dans un fichier, il faut qu'il soit dans « l'état d'écriture ».

L'instruction dont le schéma suit permet de faire passer

de lecture en écriture l'état d'un fichier ouvert par une instruction FILES :

SCRATCH ≠ indicateur

L'indicateur est une expression quelconque.

Exemple :

100 SCRATCH ≠ 2 × I + 1

La valeur V de cette expression doit cependant être un nombre entier positif et au plus égal au nombre de fichiers qui ont été ouverts par des instructions FILES. En effet, la valeur V permet de sélectionner un fichier dans la liste des fichiers ouverts : *celui dont le numéro d'ordre est égal à V.*

L'effet de l'instruction SCRATCH est de :

— mettre le fichier sélectionné par l'indicateur dans l'état d'écriture ;
— repositionner le pointeur de ce fichier sur la première information (s'il n'y était pas déjà).

Après cette instruction, il sera donc possible d'écrire des informations dans ce fichier. Remarquons cependant que, comme le pointeur est repositionné au début du fichier, la première écriture qui suit une instruction SCRATCH, se fera dans la première position du fichier.

L'opération inverse est aussi nécessaire, car comme nous l'avons dit, il est impossible de lire dans un fichier qui est dans l'état d'écriture. Elle est réalisée par l'instruction dont le schéma est (1) :

RESTØRE ≠ indicateur.

Elle est très analogue à l'instruction SCRATCH, à ceci près qu'elle met dans « l'état de lecture », le fichier sélectionné par l'indicateur.

(1) Dans les instructions SCRATCH et RESTØRE, ainsi que dans celles qui suivent, le symbole ≠ indique que le fichier utilisé est séquentiel.

L'écriture d'informations dans un fichier est réalisée au moyen d'une instruction de la forme suivante :

WRITE # indicateur, liste d'expressions.

Les expressions de la liste peuvent être des chaînes ou des variables de chaîne. Elles doivent être séparées par des points-virgule.

Exemple :

```
100 WRITE # 1, 4 × SQR(X ↑ 2 — 1) ; R$ ; Y ; " FIN "
```

Le fichier sélectionné par l'indicateur doit être dans l'état d'écriture. Il est par exemple interdit d'écrire :

```
10 FILES STØCK
20 WRITE # 1, 104
```

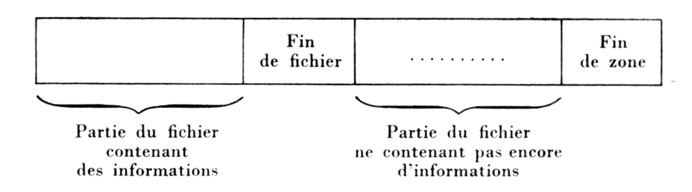
Par contre, on écrira :

```
10 FILES STØCK
15 SCRATCH # 1
20 WRITE # 1, 104
```

Au moment de la préparation du fichier sélectionné par l'indicateur, une certaine zone, caractérisée par un certain nombre de « cases » lui a été réservée sur le périphérique. Dans ces « cases », on placera les différentes informations du fichier. Le nombre de cases réservées dépend du fichier. Il existe dans le fichier une case spéciale qui en indique la fin physique, c'est-à-dire la dernière case *réservée* : elle est appelée *fin de zone* du fichier.

En outre, des informations ont éventuellement été écrites dans le fichier, au moment de sa préparation ; cependant, toutes les cases réservées n'ont pas nécessairement été remplies. Lors de la lecture dans le fichier (voir ci-dessous), il ne peut être question de lire des informations « au-delà » de ce qui a été écrit. C'est pourquoi, il existe une autre case spéciale dans le fichier, située immédiatement après celle où figure la dernière information écrite. Elle indique l'endroit extrême jusqu'où des infor-

mations ont été écrites précédemment dans le fichier : elle est appelée *fin de fichier*.



Le numéro d'ordre de la case « fin de fichier » est au plus égal à celui de la case « fin de zone ».

L'effet d'une instruction WRITE est tel qu'après son exécution, le pointeur du fichier repère toujours la *fin du fichier* (voir ci-dessous). Il en est donc de même au début de l'exécution de cette instruction sauf si, *immédiatement avant*, une instruction SCRATCH a été exécutée, auquel cas le pointeur repère la première case du fichier (qui peut ou non contenir la fin de fichier).

Donc, lors de l'exécution de l'instruction WRITE, l'ordinateur

— sélectionne un fichier au moyen de l'indicateur figurant dans l'instruction ;
— efface le cas échéant la fin de fichier, si le pointeur de ce fichier en repère une à ce moment ;
— calcule la valeur V1 de la première expression dans la liste figurant dans l'instruction ;
— écrit la valeur V1 dans la case repérée à ce moment par le pointeur ;
— déplace ce pointeur d'une case vers la droite ;
— calcule la valeur V2 de la seconde expression de la liste, si elle existe ;
— écrit cette valeur V2, dans la case repérée à ce moment par le pointeur, c'est-à-dire immédiatement après V1 ;
— déplace le pointeur d'une case vers la droite, et ainsi de suite jusqu'à épuisement des expressions de la liste ;
— met enfin une marque de fin de fichier à l'endroit où se trouve le pointeur après les écritures, *sans déplacer* ce pointeur.

Remarquons qu'après l'exécution de cette instruction, le fichier reste dans l'état d'écriture et il y restera jusqu'à ce que l'ordinateur exécute une instruction RESTØRE.

Exemple :

```
10  FILES F
20  SCRATCH # 1
30  LET I = 1
40  WRITE # 1, I ; 2 * I ; 3 * I
50  LET I = 10
60  WRITE # 1, I ; 2 * I
```

Après exécution des instructions aux lignes 10 à 40, le fichier F contient 1 dans sa première case, 2 dans la seconde, 3 dans la troisième et la marque de fin de fichier dans la quatrième. Le pointeur repère alors cette quatrième position. Lors de l'exécution de l'instruction à la ligne 60, la marque de fin de fichier est effacée, le nombre 10 est écrit à sa place, puis le nombre 20 est écrit dans la cinquième case du fichier, enfin une marque de fin de fichier est placée dans la sixième position, qui est celle que le pointeur repère alors.

Pour éviter une écriture (ou une lecture) intempestive hors des limites d'un fichier, il est possible d'en tester la fin, au moyen d'une instruction de la forme :

IF END # indicateur THEN numéro de ligne.

L'effet de cette instruction est le suivant :

— lorsque le fichier sélectionné par l'indicateur est dans « l'état d'écriture », l'ordinateur teste si le pointeur de ce fichier repère actuellement la fin de zone ; lorsque ce fichier est dans l'état de lecture, il teste si le pointeur repère actuellement la fin de zone *ou* la fin de fichier ;
— si le test précédent est positif, l'ordinateur exécute ensuite l'instruction dont le numéro de ligne est indiqué après THEN. Sinon, il exécute l'instruction qui suit immédiatement le test (comme pour une instruction conditionnelle).

La lecture d'informations dans un fichier se fait au moyen d'une instruction de la forme suivante :

READ # indicateur, liste de variables.

Les variables de la liste doivent être séparées par des virgules.

Exemple :

```
100  READ # 4, X, Y, Z, R$, U
```

Le fichier sélectionné par l'indicateur doit être dans l' « état de lecture ».

Exemple : Les deux lignes suivantes sont correctes :

```
10  FILES STØCK
20  READ # 1, X$, U, V
```

Par contre, les lignes suivantes sont incorrectes :

```
10  FILES STØCK
20  SCRATCH # 1
30  WRITE # 1, " BØULØN "
40  READ # 1, X$, U, V
```

En effet, après la ligne 20, le fichier STØCK est dans « l'état d'écriture » et y reste après exécution de l'instruction à la ligne 30.

Pour que les lignes précédentes deviennent correctes, il faut remettre le fichier STØCK dans « l'état de lecture » avant la ligne 40, au moyen de l'instruction RESTØRE.

Lors de l'exécution d'une instruction READ, l'ordinateur lit la valeur (nombre ou chaîne) repérée à ce moment par le pointeur du fichier sélectionné par l'indicateur, l'affecte à la première variable de la liste, déplace le pointeur d'une case vers la droite et recommence ce même processus pour les variables suivantes de la liste, jusqu'à épuisement de ces variables.

Exemple :

```
10  FILES F
20  SCRATCH # 1
```

```
30 WRITE # 1, 1 ; 2 ; 3
40 RESTØRE # 1
50 READ # 1, X, Y, Z
```

Après exécution de ces instructions, les valeurs des variables X, Y et Z sont respectivement 1, 2 et 3 ; (ces valeurs ont été écrites à la ligne 30 dans les trois premières positions du fichier F).

Remarques :

1) Après exécution d'une instruction READ, le fichier reste dans l'état de lecture et y restera jusqu'à exécution d'une instruction SCRATCH. De plus, le pointeur repère la case qui suit immédiatement celle qui contient la dernière valeur lue dans le fichier.

2) L'ordinateur signale une erreur si, au moment d'une lecture dans un fichier, le pointeur repère la fin de ce fichier. Il en signale aussi une, si la valeur lue n'est pas du même type (nombre ou chaîne) que la variable à laquelle elle doit être affectée.

Exemple :

```
10 FILES F
20 SCRATCH # 1
30 WRITE # 1, " FIN "
40 RESTØRE # 1
50 READ # 1, X
```

L'ordinateur signalera une erreur à la ligne 50 car la valeur lue dans le fichier F est la chaîne FIN (écrite à la ligne 30), qui devrait être affectée à la variable simple X.

Lorsqu'un fichier est dans l'état de lecture, il est possible de ramener son pointeur d'une position vers la gauche, au moyen d'une instruction de la forme suivante :

BACKSPACE # indicateur.

L'ordinateur signale une erreur si, au moment de l'exécution de cette instruction, le pointeur repère la première position du fichier sélectionné par l'indicateur.

Cette instruction peut être répétée pour ramener le pointeur d'un fichier de plusieurs positions vers la gauche.

Exemple : Dans le programme suivant, les lignes 20 à 60 permettent de lire un fichier F du début à la fin, en comptant le nombre N d'informations contenues dans F. La boucle POUR des lignes 70 à 150 lit les informations du fichier F *en sens inverse* et les utilise dans un « traitement » qui n'est pas détaillé. Le fichier F est supposé contenir uniquement des nombres (au moins un).

```
 10  FILES F
 20  LET N = 0
 30  READ # 1, X
 40  LET N = N + 1
 50  IF END # 1 THEN 70
 60  GØTØ 30
 70  FØR I = 1 TØ N
 80  BACKSPACE # 1
 90  READ # 1, X
100  BACKSPACE # 1
     « traitement »
150  NEXT I.
```

Lors de la première exécution de l'instruction à la ligne 80, le pointeur de F repère la position N + 1, occupée par la fin de fichier (ou de zone). Après exécution de cette instruction, le pointeur repère donc la N-ième position de F et la ligne 90 permet de lire la N-ième information (donc la dernière) de F. A ce moment, le pointeur de F repère de nouveau la N + 1-ième position, mais la ligne 100 ramène le pointeur à la position N. Ainsi, après le traitement et lors de la seconde exécution de la boucle, la ligne 80 aura pour effet de ramener le pointeur à la position N — 1 et la ligne 90 permettra alors de lire l'avant-dernière position de F, et ainsi de suite jusqu'à la première information de F.

Seconde forme : L'ouverture de fichiers se fait au moyen de l'instruction suivante :

ØPEN nom d'un fichier, état.

L'état est spécifié par un des deux mots :

INPUT
ØUTPUT

Exemple :

```
10  ØPEN STØCK, INPUT
 .
 .
 .
50  ØPEN STØCK, ØUTPUT
```

Contrairement à l'instruction FILES de la première forme, l'instruction ØPEN peut être écrite à un endroit quelconque du programme.

Le nom précisé après ØPEN doit correspondre à un fichier préalablement préparé.

L'effet de l'instruction ØPEN est double :

— elle positionne le pointeur du fichier sur la première information. Ce sera donc la seule immédiatement accessible après cette instruction ;
— elle place le fichier dans l'état spécifié :
    — si l'état est spécifié par INPUT, le fichier passe dans l'état de lecture. On peut donc y lire mais non y écrire des informations ;
    — s'il est spécifié par ØUTPUT, le fichier passe dans l'état d'écriture : on peut y écrire mais non y lire des informations.

Remarques :

1) Sous réserve des restrictions indiquées ci-dessous, cette instruction permet de faire passer un fichier F de l'état de lecture dans l'état d'écriture, ou l'inverse.

2) L'ouverture d'un nouveau fichier dans un certain état, oblitère les ouvertures des *autres* fichiers dans le même état à ce moment. Pour utiliser l'un de ces derniers dans ce même état, il faudra donc le réouvrir au moyen d'une instruction ØPEN.

Exemple :

```
10  ØPEN STØCK, INPUT
20  ØPEN CLIENTELE, ØUTPUT
30  ØPEN MAGASINS, INPUT
40  ØPEN STØCK, ØUTPUT
```

A la ligne 30, l'ouverture en lecture du fichier MAGASINS oblitère l'ouverture en lecture du fichier STØCK faite à la ligne 10. Ainsi, après la ligne 30 on ne pourra lire d'informations dans le fichier STØCK, que si on l'ouvre de nouveau en lecture.

Par contre, l'ouverture *en lecture* du fichier MAGASINS à la ligne 30 n'oblitère pas celle en écriture du fichier CLIENTELE, faite à la ligne 20, car ces deux fichiers ne sont pas dans le même état. Après la ligne 30, le fichier CLIENTELE est toujours dans l'état d'écriture mais il le perdra après la ligne 40 où le fichier STØCK est ouvert dans l'état d'écriture.

Ainsi, à tout moment, il y a au plus un fichier ouvert dans l'état de lecture et un autre, *différent*, ouvert dans l'état d'écriture.

3) L'ouverture d'un fichier dans l'état de lecture est interdite si ce fichier est déjà dans l'état d'écriture, et inversement.

Exemple :

Les deux lignes suivantes sont incorrectes :

```
10  ØPEN STØCK, INPUT
20  ØPEN STØCK, ØUTPUT
```

de même que les deux lignes :

```
10  ØPEN STØCK, ØUTPUT
20  ØPEN STØCK, INPUT
```

Pour pouvoir ouvrir dans l'état d'écriture un fichier se trouvant dans l'état de lecture, il est nécessaire d'ouvrir préalablement un *autre* fichier dans l'état de lecture, ce qui oblitère l'ouverture du premier fichier. Il en est de même pour faire passer un fichier de l'état d'écriture dans l'état de lecture.

Exemple :

```
10  ØPEN STØCK, INPUT
20  ØPEN AUX, INPUT
```

```
30  ØPEN STØCK, ØUTPUT
40  ØPEN AUX 1, ØUTPUT
50  ØPEN STØCK, INPUT
```

L'ouverture à la ligne 20 du fichier AUX dans l'état de lecture oblitère celle du fichier STØCK faite à la ligne 10. Le fichier STØCK peut alors être ouvert dans l'état d'écriture (ligne 30). De même, l'ouverture à la ligne 40 du fichier AUX 1 dans l'état d'écriture oblitère celle du fichier STØCK faite à la ligne 30. Ce dernier peut alors être de nouveau ouvert dans l'état de lecture (ligne 50).

Dans la seconde forme, l'écriture d'informations sur un fichier se fait au moyen de l'instruction suivante :

PRINT FILE liste d'expressions entre virgules.

Exemple :

```
100  PRINT FILE X — Y / Z, SQR(X), 3, X * Y
```

Cette instruction écrit les valeurs des expressions de la liste dans un fichier de manière analogue à l'instruction WRITE de la première forme, à ceci près qu'il n'existe pas de marque de fin de fichier. (En particulier, l'utilisateur devra mettre lui-même un symbole spécial à la fin du fichier pour savoir, le cas échéant, où son fichier se termine. Il n'existe pas non plus de fin de zone ni de test de fin de fichier (ou de zone).)

Le fichier sur lequel sont écrites les valeurs est le dernier qui ait été ouvert dans « l'état d'écriture » avant l'exécution de cette instruction. (Rappelons qu'il existe toujours au plus un fichier ouvert dans l'état d'écriture.)

Exemple :

```
10  ØPEN F, ØUTPUT
20  PRINT FILE 1, 2, 3
```

Ces deux lignes ont pour effet d'écrire les valeurs 1, 2 et 3 dans les trois *premières* (voir l'instruction ØPEN) positions du fichier F, ouvert dans l'état d'écriture à la ligne 10.

La lecture d'informations sur un fichier se fait au moyen de l'instruction suivante :

READ FILE liste de variables.

Son effet est tout à fait analogue à celui de l'instruction READ étudiée pour la première forme.

Le fichier sur lequel les informations sont lues est le dernier ayant été ouvert dans l'état de lecture.

Exemple :

```
10  ØPEN F, ØUTPUT
20  PRINT FILE 1, 2, 3
30  ØPEN AUX, ØUTPUT
40  ØPEN F, INPUT
50  READ FILE X, Y, Z
```

Après la ligne 50, les variables X, Y et Z ont respectivement comme valeurs 1, 2 et 3. Ces valeurs ont été écrites dans les trois premières positions du fichier F à la ligne 20.

Exemple d'utilisation d'un fichier séquentiel. La modification d'une ou de plusieurs informations d'un fichier séquentiel exige l'utilisation d'un fichier auxiliaire. Pour le montrer, prenons un exemple simple (1) : il s'agit de modifier la dixième information d'un fichier F, supposé ne contenir que des nombres et de remplacer cette valeur par 1.

Cela peut être réalisé au moyen d'une instruction WRITE pourvu qu'au moment de l'exécution de cette instruction :

— le pointeur du fichier F repère la dixième position ;
— le fichier F soit dans « l'état d'écriture ».

La première condition ne peut être réalisée que si l'on a effectué d'abord neuf lectures (celles des neuf premières informations) ou neuf écritures dans le fichier F. Dans le premier cas, le fichier F sera dans l'état de lecture lorsque son pointeur repérera la dixième position. On ne pourra donc pas y écrire directement la valeur 1. De plus, le fait de faire passer à ce

(1) Nous écrirons les programmes en utilisant les instructions de la première forme. Il en serait de même avec celle de la seconde forme.

moment le fichier de l'état de lecture à l'état d'écriture ramène le pointeur sur la première information, ce qui annule tout le travail précédent, dont le seul but était de déplacer le pointeur.

La solution d'effectuer neuf écritures dans F est aussi impossible. En effet, pour éviter de détruire les anciennes informations du fichier F, il faudrait pouvoir les connaître afin d'être sûr d'écrire les « bonnes » informations aux neuf premières places du fichier F. Ceci ne peut se faire qu'en lisant ces informations et nous venons de voir que cette solution est impossible.

La seule possibilité restant est de constituer un fichier auxiliaire AUX, dans lequel on écrira les anciennes informations de F, sauf la dixième qui sera remplacée par 1. Après cette opération, le fichier AUX contiendra les informations modifiées qu'on désire mettre dans le fichier F : il suffira alors de recopier le fichier AUX dans le fichier F.

Le programme suivant réalise ce travail (la variable N sert à compter le nombre d'informations du fichier F, afin d'éviter un test de fin de fichier lors de la recopie).

```
 10  FILES F ; AUX
 20  SCRATCH # 2
 30  FØR I = 1 TØ 9
 40  READ # 1, X
 50  WRITE # 2, X
 60  NEXT I
 70  READ # 1, X
 80  WRITE # 2, 1
 90  LET N = 10
100  IF END # 1 THEN 150
110  READ # 1, X
120  WRITE # 2, X
130  LET N = N + 1
140  GØTØ 100
150  RESTØRE # 2
160  SCRATCH # 1
170  FØR I = 1 TØ N
180  READ # 2, X
190  WRITE # 1, X
```

```
200 NEXT I
210 END
```

Les lignes 20 à 60 réécrivent les neuf premières informations du fichier F dans le fichier AUX.

La ligne 70 « saute » la dixième information de F (qui ne doit pas être réécrite dans AUX) et la ligne 80 écrit la valeur 1 comme dixième information du fichier AUX.

Les lignes 100 à 140 recopient les autres informations du fichier F sur le fichier AUX, à partir de la onzième. Enfin, les lignes 150 à 200 recopient le fichier AUX sur le fichier F.

XI.4. Fichiers a accès direct. Utilisation en BASIC

Les fichiers séquentiels présentent plusieurs inconvénients :

— la recherche d'une information dans le fichier oblige à déplacer le pointeur depuis le début jusqu'à la case contenant cette information, au moyen d'instructions de lecture. Celles-ci ne servent qu'à déplacer le pointeur. Or, elles sont souvent coûteuses en temps ;

— la modification d'une information dans le fichier oblige à recopier deux fois ce fichier, comme nous l'avons vu dans l'exemple à la fin du paragraphe précédent.

Dans les deux cas, le temps de traitement est disproportionné avec son volume (accès ou modification d'une seule information). Comme nous l'avons dit, il ne devient acceptable que si le nombre d'informations auxquelles on veut accéder, ou que l'on veut modifier, est suffisant. Si ce n'est pas le cas, il est préférable d'utiliser l'autre type de fichier : le fichier à accès direct.

Dans ce cas, il est possible de positionner directement le pointeur à n'importe quel endroit du fichier (voir ci-dessous). De plus, les lectures et écritures dans le fichier peuvent être alternées. Pour modifier une information d'un fichier, il ne sera donc plus nécessaire d'utiliser un fichier auxiliaire.

Remarquons cependant que la facilité procurée par les

fichiers à accès direct n'est utilisable que dans la mesure où l'on peut calculer aisément le numéro d'ordre d'un élément du fichier. C'est le cas si l'on connaît une information caractéristique de cet élément, telle que sa désignation ou son code. En cas contraire, il est impossible de positionner le pointeur du fichier sur l'information désirée.

En BASIC, l'utilisation des fichiers à accès direct n'est pas possible sous la seconde forme décrite au paragraphe précédent. Sous la première forme, les instructions sont très analogues à celles qui permettent d'utiliser un fichier séquentiel.

En particulier, il n'existe aucune différence entre l'ouverture d'un fichier séquentiel et celle d'un fichier à accès direct : elles se font toutes deux au moyen de l'instruction FILES décrite plus haut. Un fichier ouvert par une instruction FILES peut donc être aussi bien séquentiel qu'à accès direct. Le type du fichier sera déterminé en fonction de l'utilisation qui en sera faite dans le programme. Remarquons que si un fichier est utilisé en tant que fichier séquentiel dans un programme, il ne pourra être utilisé ultérieurement en tant que fichier à accès direct, et réciproquement.

Il est possible de positionner le pointeur d'un fichier à accès direct, à n'importe quel endroit de ce fichier, au moyen de l'instruction (1) :

SET : indicateur, expression.

L'indicateur y joue le même rôle que pour les fichiers séquentiels.

L'effet de cette instruction est le suivant : l'ordinateur calcule la valeur V de l'expression et positionne le pointeur du fichier sélectionné par l'indicateur à la V-ième position. Il est bien sûr nécessaire que la valeur V soit un nombre entier positif et au plus égal aux nombres de cases réservées pour le fichier lors de sa préparation.

Exemple :

```
100   SET : 2, 4 * N + J
```

(1) Le symbole : indique que le fichier est utilisé en tant que fichier à accès direct.

Les instructions d'écriture et de lecture des informations sont analogues à celles écrites pour les fichiers séquentiels. Le symbole # est remplacé par le symbole : . Leur fonctionnement est identique à ceci près qu'un fichier à accès direct ne possède pas d'état de lecture ou d'écriture. Il est donc possible de lire une information dans un fichier à accès direct après en avoir écrit une et ceci sans recommencer au début du fichier.

Exemple :

```
 10  FILES F1, F2
  .
  .
  .
100  SET : 1, 10
110  READ : 1, X
120  WRITE : 1, 2
```

La dixième information du fichier 1 est lue à la ligne 110 et affectée à X. La valeur 2 est écrite immédiatement après, c'est-à-dire dans la onzième position du fichier 1 (lors de l'exécution de la ligne 110, le pointeur s'est déplacé d'une position vers la droite).

Le programme décrit à la fin du § XI.3 pourra s'écrire de manière beaucoup plus simple avec un fichier à accès direct :

```
10  FILES F
20  SET : 1, 10
30  WRITE : 1, 1
40  END
```

Les instructions SCRATCH et RESTØRE n'ont plus d'utilité pour les fichiers à accès direct. Elles existent cependant (en remplaçant le symbole # par :) et leur seul effet est de placer le pointeur en première position du fichier.

L'instruction qui permet de tester la fin de fichier ou de zone existe aussi pour les fichiers à accès direct (# y est remplacé par :). Son effet est le même que pour les fichiers séquentiels.

Enfin, deux fonctions standards donnent des renseignements importants pour les fichiers à accès direct :

La première de sigle LØC donne la position du pointeur du fichier sélectionné par la valeur du paramètre.

Exemple :

```
10  FILES F1, F2, F3
20  LET I = 1
30  READ : 1, X
40  LET J = LØC (I)
50  LET I = I + 1
60  READ : I, Y, Z
70  LET K = LØC (I)
 .
 .
 .
```

Après exécution des instructions des lignes 10 à 70, la valeur de J est 2 (le pointeur du fichier F1 est positionné sur la deuxième case) et celle de K est 3.

La seconde fonction de sigle LØF donne la position de la fin de zone pour le fichier sélectionné par la valeur du paramètre. Ainsi le test :

LØC(1) = LØF(1)

permet de savoir si le pointeur du premier fichier est positionné ou non à la fin de ce fichier.

Exemple :

```
 10  FILES F
  .
  .
  .
100  IF LØC(1) < > LØF(1) THEN 130
110  PRINT " FIN DU FICHIER F "
120  STØP
  .
  .
  .
```

EXERCICES

— Qu'entend-on par fichier ? Quels sont les périphériques utilisés pour stocker une grande quantité d'informations ?

— Quels sont les types de traitement effectués sur un fichier ?

— Quelle différence faites-vous entre un fichier à accès direct et un fichier séquentiel ?

— Dans un même programme, pouvez-vous utiliser des fichiers à accès direct et des fichiers séquentiels ?

— Un de vos fichiers étant actuellement en état de lecture, de quelle(s) instruction(s) disposez-vous pour le mettre en état d'écriture ?

— Pour un fichier à accès direct, quelle est l'instruction qui permet de placer le pointeur sur l'enregistrement à lire. Comment feriez-vous pour écrire une nouvelle information à la place de l'élément qui vient d'être lu ?

— Dans l'exercice de calcul d'impôt qui se trouve à la fin du chapitre VII :
— lire les revenus du couple et les écrire dans un fichier F1 ;
— dans le cas où le couple paie un impôt, calculer la somme des revenus et l'écrire dans un fichier F2.

L'exercice sera donc repris à partir de données se trouvant dans les fichiers F1 et F2.

— Dans l'exercice de dénombrement statistique qui se trouve à la fin du chapitre IX :

— enregistrer les données sur un support magnétique ;
— ranger les classes et les effectifs par classe dans un autre fichier.

BIBLIOGRAPHIE

[1] J. Arsac, *Les systèmes de conduite des ordinateurs*, Dunod.

[2] J. Arsac, *La science informatique*, Dunod.

[3] V. Cordonnier, *La gestion des fichiers en informatique*, Presses Universitaires de France (à paraître).

[4] P. Demarne, M. Rouquerol, *Les ordinateurs électroniques*, Presses Universitaires de France (coll. « Que sais-je ? », nº 832).

[5] J. du Roscoät, *Conception de la programmation des ordinateurs*, Masson.

[6] J. Kemeny, T. E. Kurtz, *Basic programming*, John Wiley and Sons.

[7] P. Poulain, *Eléments fondamentaux de l'informatique*, Dunod.

TABLE DES MATIÈRES

AVANT-PROPOS 5

CHAPITRE PREMIER. — **L'ordinateur et ses modes de fonctionnement** 7

I.1. L'ordinateur 7
I.2. Fonctionnement d'un ordinateur 10
I.3. Les différentes utilisations d'un ordinateur 13

CHAPITRE II. — **Introduction au langage BASIC**

II.1. Exemple de programme 19
II.2. Les objets manipulés en BASIC 24

CHAPITRE III. — **Les calculs en BASIC. Instruction d'affectation** 29

III.1. Les expressions arithmétiques 29
III.2. Ordre des opérations dans une expression 30
III.3. Parenthèsage des expressions 33
III.4. Les appels de fonctions 35
III.5. L'instruction d'affectation 39

CHAPITRE IV. — **Sorties des résultats. Instruction PRINT** 42

IV.1. Impression d'une ou de plusieurs valeurs 42
IV.2. Impression de textes : notion de chaîne 46
IV.3. Impression condensée 48
IV.4. Tabulation 49

CHAPITRE V. — **Notion d'algorithme. Entrée des données en BASIC** 52

V.1. Calcul symbolique et algorithme 52
V.2. Analyse d'un problème. Organigramme 55
V.3. Entrée des données en BASIC 57

CHAPITRE VI. — **Boucles de calcul et ruptures de séquences** 62

VI.1. Ruptures de séquences inconditionnelles 63
VI.2. Ruptures de séquences calculées 67

CHAPITRE VII. — **Calculs itératifs** 73

VII.1. Rupture de séquence conditionnelle 73
VII.2. Instruction conditionnelle et instruction GØTØ 76
VII.3. Cas de tests successifs 82
VII.4. Contrôle de boucle. Boucle POUR 86
VII.5. Boucles imbriquées 93

Chapitre VIII. — **Entrées des données à partir du terminal. Variables alphanumériques** 97

VIII.1. Instruction INPUT 98
VIII.2. Variables alphanumériques 102
VIII.3. Calculs sur les variables alphanumériques 104

Chapitre IX. — **Les listes et les tables en BASIC** 108

IX.1. Description des variables de liste 109
IX.2. Utilisation d'une variable de liste 112
IX.3. Variables de tableau 116
IX.4. Utilisation des variables de tableau 119
IX.5. Instructions matricielles 123

Chapitre X. — **Les sous-programmes** 131

X.1. Sous-programmes GØSUB 132
X.2. Sous-programmes CALL 140

Chapitre XI. — **Les fichiers** 145

XI.1. Utilisation des fichiers 146
XI.2. Les fichiers séquentiels 147
XI.3. Utilisation des fichiers séquentiels en BASIC 148
XI.4. Fichiers à accès direct. Utilisation en BASIC 163

Bibliographie 169

1973. — Imprimerie des Presses Universitaires de France. — Vendôme (France)
ÉDIT. N° 32 554 IMPRIMÉ EN FRANCE IMP. N° 23 398

THÉMIS

La collection *Thémis* a pour premier but de fournir aux étudiants des moyens de travail modernes. Elle a précédé sur ce point les réformes universitaires déclenchées en 1968, tant par la structure de ses manuels que par leur domaine.

Les manuels *Thémis* comportent deux séries de développements distincts : 1° l'une, en caractères normaux, permet aux étudiants d'avoir une vue d'ensemble de la matière, dont les cours magistraux ne développent que certains aspects ; 2° l'autre, en petits caractères, comprend à la fois des « états des questions » analysant les principaux problèmes controversés et un guide très étendu de bibliographie commentée : ainsi l'étudiant peut approfondir les questions spécialement traitées dans le cours et participer efficacement aux travaux dirigés. A ce dernier point de vue, une série complémentaire de « Textes et Documents » met à sa disposition, sous une forme commode, un ensemble de documents constamment tenus à jour, dont beaucoup sont inédits ou difficilement accessibles.

Par ailleurs, la collection *Thémis* a toujours embrassé tous les aspects des sciences sociales, sans tenir compte des cloisonnements artificiels établis par les institutions universitaires traditionnelles. Dès l'origine, elle a englobé des manuels juridiques, économiques et politiques malgré les divisions existant alors entre les Facultés de Droit, les Facultés des Lettres, les Instituts d'Etudes politiques, etc. L'affaiblissement de ces divisions permet maintenant d'appliquer intégralement le programme initial de la collection, de lui donner un caractère entièrement interdisciplinaire et de supprimer toute distinction par années de licence ou par établissements universitaires.

Le développement de l'autonomie des Universités permet également d'accroître une liberté de présentation limitée jusqu'ici par la rigueur de programmes uniformes et impératifs. Au lieu de correspondre chacun à l'ensemble d'un enseignement — souvent différent désormais d'une Faculté à une autre — les nouveaux manuels *Thémis* seront en général découpés en ouvrages plus restreints, pouvant être regroupés de façon souple pour correspondre à la diversité des cours suivant les Universités.

THÉMIS

SECTION DROIT

1 **Droit civil, 1 :** Introduction, les Personnes / Jean CARBONNIER
2 **Droit civil, 2 :** La Famille, les Incapacités / Jean CARBONNIER
3 **Droit civil, 3 :** Les Biens / Jean CARBONNIER
4 **Droit civil, 4 :** Les Obligations / Jean CARBONNIER
5 **Les régimes matrimoniaux** / Gérard CORNU
6 **Droit administratif** / Georges VEDEL
7 **Procédure civile** / Gérard CORNU et Jean FOYER
8 **Sécurité sociale** / Jacques DOUBLET
9 **Histoire du droit privé, 1 :** Les Obligations / Paul OURLIAC et J. de MALAFOSSE
10 **Histoire du droit privé, 2 :** Les Biens / Paul OURLIAC et J. de MALAFOSSE
11 **Histoire du droit privé, 3 :** Le Droit familial / Paul OURLIAC et J. de MALAFOSSE
12 **Droit du travail** / Jean RIVERO et Jean SAVATIER
13 **Droit aérien** / Louis CARTOU
14 **Droit international public** / Paul REUTER
15 **Droit d'outre-mer et de la coopération** / François LUCHAIRE
16 **Eléments de droit public** / Maurice DUVERGER
17 **Droit privé civil et commercial, 1** / Robert VOUIN et Pierre ROBINO
18 **Droit privé civil et commercial, 2** / Robert VOUIN et Pierre ROBINO
19 **Procédure civile et voies d'exécution** / Pierre CATALA et François TERRÉ
20 **Droit administratif spécial** / André de LAUBADÈRE
21 **Droit pénal et procédure pénale** / Robert VOUIN et Jacques LÉAUTÉ
22 **Grands services publics et entreprises nationales, 1** / Jean-Marie AUBY et Robert DUCOS-ADER
23 **Grands services publics et entreprises nationales, 2** / Jean-Marie AUBY et Robert DUCOS-ADER
24 **Droit commercial, 1 :** Introduction, les Entreprises / Paul DIDIER
25 **Droit économique** / Gérard FARJAT
26 **Criminologie et science pénitentiaire** / Jacques LÉAUTÉ
27 **Droit rural** / Raymond MALÉZIEUX

SECTION GESTION

1 **Gestion de l'entreprise, 1 :** Structure et organisation / Jane AUBERT-KRIER
2 **Gestion de l'entreprise, 2 :** Activités et politiques / Jane AUBERT-KRIER, E.-Y. RIO et Ch.-A. VAILHEN
3 **Comptabilité privée** / Jane AUBERT-KRIER
4 **Informatique** / Vincent CORDONNIER
5 **Le langage BASIC** / Baudouin DRIEUX et André-Louis LIJU
6 **Politique des firmes industrielles** / Jean PARENT
7 **Gestion des fichiers** / Vincent CORDONNIER
8 **Langages de programmation** / Baudouin DRIEUX et Christian CARREZ
9 **Marketing** / Georges FRÊCHE

THÉMIS

SECTION SCIENCES ÉCONOMIQUES

1 **Economie politique, 1** / Raymond BARRE
2 **Economie politique, 2** / Raymond BARRE
3 **Mathématiques préparatoires à l'économie, 1** / Georges Th. GUILBAUD
4 **Statistique, 1 :** Statistique descriptive et initiation à l'analyse / André PIATIER
5 **Histoire des faits économiques contemporains** / Maurice NIVEAU
6 **Histoire de la pensée économique** / Henri DENIS
7 **Economie internationale** / Gérard MARCY
8 **Economie financière** / Hubert BROCHIER et Pierre TABATONI
9 **Histoire économique, des origines à 1789** / Jean IMBERT et Henri LEGOHÉREL
10 **Systèmes et structures économiques** / André MARCHAL
11 **Economie rurale** / Jules MILHAU et Roger MONTAGNE
12 **Economie du travail** / François SELLIER et André TIANO
13 **Eléments d'économie politique** / Jean-Marcel JEANNENEY
14 **Statistique et observation économique** / André PIATIER
15 **Comptabilité nationale et modèles de politique économique** / Jean BÉNARD
16 **Eléments d'économétrie** / René ROY
17 **Economie européenne** / Jacques et Colette NÊME
18 **Modèles économiques** / Marc GUILLAUME
19 **Economie contemporaine, 1 :** Les fonctions économiques / Denise FLOUZAT
20 **Organisations économiques internationales** / Jacques et Colette NÊME

SECTION SCIENCES POLITIQUES

1 **Sociologie politique** / Maurice DUVERGER
2 **Institutions internationales** / Paul REUTER
3 **Histoire des Institutions, 1-2 :** L'Antiquité / Jacques ELLUL
4 **Histoire des Institutions, 3 :** Le Moyen Age / Jacques ELLUL
5 **Histoire des Institutions, 4 :** XVIe-XVIIIe siècle / Jacques ELLUL
6 **Histoire des Institutions, 5 :** Le XIXe siècle / Jacques ELLUL
7 **Finances publiques** / Maurice DUVERGER
8 **Histoire des idées politiques, 1 :** Des origines au XVIIIe siècle / Jean TOUCHARD, Louis BODIN, Pierre JEANNIN, Georges LAVAU et Jean SIRINELLI
9 **Histoire des idées politiques, 2 :** Du XVIIIe siècle à nos jours / Jean TOUCHARD, Louis BODIN, Pierre JEANNIN, Georges LAVAU et Jean SIRINELLI
10 **Organisations européennes** / Paul REUTER
11 **Administration publique** / Bernard GOURNAY, Jean-François KESLER et Jeanne SIWEK-POUYDESSEAU
12 **Amérique latine** / Jacques LAMBERT
13 **La vie politique en France depuis 1940** / Jacques CHAPSAL

THÉMIS

SECTION SCIENCES POLITIQUES (suite)

14 **Les régimes parlementaires européens** / Pierre LALUMIÈRE et André DEMICHEL
15 **Les régimes politiques des pays arabes** / Maurice FLORY et Robert MANTRAN
16 **Histoire de l'Administration** / Pierre LEGENDRE
17 **Science fiscale** / Lucien MEHL
18 **Les régimes politiques de l'U.R.S.S. et de l'Europe de l'Est** / Michel LESAGE
19 **Les syndicats ouvriers** / Guy CAIRE
20 **Institutions politiques et droit constitutionnel, 1 :** Les grands systèmes politiques / Maurice DUVERGER
21 **Institutions politiques et droit constitutionnel, 2 :** Le système politique français / Maurice DUVERGER
22 **Les dictatures européennes** / André et Francine DEMICHEL

SECTION SCIENCES SOCIALES

1 **Mathématiques élémentaires. Applications à la statistique et aux sciences sociales, 1** / Claude d'ADHÉMAR, Marc BARBUT, Pierre JULLIEN et Bruno LECLERC
2 **Méthodes des sciences sociales** / Maurice DUVERGER
3 **Démographie** / Philippe MOUCHEZ

SECTION TEXTES ET DOCUMENTS

1 **Constitutions et documents politiques** / Maurice DUVERGER
2 **Documents économiques, 1** / Jean-Marcel JEANNENEY, Raymond BARRE, Maurice FLAMANT et Marguerite PERROT
3 **Documents économiques, 2** / Jean-Marcel JEANNENEY, Raymond BARRE, Maurice FLAMANT et Marguerite PERROT
4 **Traités et documents diplomatiques** / Paul REUTER et André GROS
5 **Histoire des institutions et des faits sociaux, 1 :** Antiquité / Jean IMBERT, Gérard SAUTEL et Marguerite BOULET-SAUTEL
6 **Histoire des institutions et des faits sociaux, 2 :** X^e^-XIXe siècle / Jean IMBERT, Gérard SAUTEL et Marguerite BOULET-SAUTEL
7 **La formation de la science économique** / Henri DENIS
8 **Economie et mathématiques (Eléments et exercices), 1 :** Les nombres. Algèbre. Analyse / André PIATIER, Pierre CAHUZAC et Lucien CHAMBADAL
9 **La pensée politique, des origines à nos jours** / Jean IMBERT, Henri MOREL et René-Jean DUPUY
10 **L'Administration, du XVIIIe siècle à nos jours** / Pierre LEGENDRE
11 **Economie et mathématiques (Eléments et exercices), 2 :** Analyse statistique et applications à l'économie / André PIATIER, Pierre CAHUZAC et Lucien CHAMBADAL
12 **Géographie des élections françaises depuis 1936** / Claude LELEU
13 **Droit commercial. Les entreprises** / Paul DIDIER
14 **Droit social européen** / Jacques Jean RIBAS, Marie-José JONCZY et Jean-Claude SÉCHÉ

THÉMIS

B. DRIEUX
A.-L. LIJU

le langage BASIC

P. U. F.



Document numérisé avec amour par :

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/